XML with Qt

Qt 3.0

Contents

XML Module

This module is part of the Qt Enterprise Edition.

Overview of the XML architecture in Qt

The XML module provides a well-formed XML parser using the SAX2 (Simple API for XML) interface plus an implementation of the DOM Level 2 (Document Object Model).

SAX is an event-based standard interface for XML parsers. The Qt interface follows the design of the SAX2 Java implementation. Its naming scheme was adapted to fit the Qt naming conventions. Details on SAX2 can be found at http://www.megginson.com/SAX/.

Support for SAX2 filters and the reader factory are under development. Furthermore the Qt implementation does not include the SAX1 compatibility classes present in the Java interface.

For an introduction to Qt's SAX2 classes see "The Qt SAX2 classes". A code example is discussed in the "tagreader walkthrough".

DOM Level 2 is a W3C Recommendation for XML interfaces that maps the constituents of an XML document to a tree structure. Details and the specification of DOM Level 2 can be found at http://www.w3.org/DOM/. More information about the DOM classes in Qt is provided in the Qt DOM classes.

Qt provides the following XML related classes:

- QDomAttr — Represents one attribute of a QDomElement
- QDomCDATASection — Represents an XML CDATA section
- QDomCharacterData — Represents a generic string in the DOM
- QDomComment — Represents an XML comment
- QDomDocument — The representation of an XML document
- QDomDocumentFragment — Tree of QDomNodes which is usually not a complete QDomDocument
- QDomDocumentType — The representation of the DTD in the document tree
- QDomElement — Represents one element in the DOM tree
- QDomEntity — Represents an XML entity
- QDomEntityReference — Represents an XML entity reference
- QDomImplementation — Information about the features of the DOM implementation
- QDomNamedNodeMap — Collection of nodes that can be accessed by name
- QDomNode — The base class for all nodes of the DOM tree
- QDomNodeList — List of QDomNode objects
- QDomNotation — Represents an XML notation
- QDomProcessingInstruction — Represents an XML processing instruction
- QDomText — Represents textual data in the parsed XML document

- QXmlAttributes — XML attributes
- QXmlContentHandler — Interface to report logical content of XML data
- QXmlDeclHandler — Interface to report declaration content of XML data
- QXmlDefaultHandler — Default implementation of all XML handler classes
- QXmlDTDHandler — Interface to report DTD content of XML data
- QXmlEntityResolver — Interface to resolve extern entities contained in XML data
- QXmlErrorHandler — Interface to report errors in XML data
- QXmlInputSource — The input data for the QXmlReader subclasses
- QXmlLexicalHandler — Interface to report lexical content of XML data
- QXmlLocator — The XML handler classes with information about the actual parsing position
- QXmlNamespaceSupport — Helper class for XML readers which want to include namespace support
- QXmlParseException — Used to report errors with the QXmlErrorHandler interface
- QXmlReader — Interface for XML readers (i.e. for SAX2 parsers)
- QXmlSimpleReader — Implementation of a simple XML reader (a SAX2 parser)

The Qt SAX2 classes

Introduction to SAX2

The SAX2 interface is an event-driven mechanism to provide the user with document information. "Event" in this context has nothing to do with the term "event" you probably know from windowing systems; it means that the parser reports certain document information while parsing the document. These reported information is referred to as "event".

To make it less abstract consider the following example:

```
To make it less abstract consider the following example:
```

Whilst reading (a SAX2 parser is usually referred to as "reader") the above document three events would be triggered:

1. A start tag occurs (`<quote>`).
2. Character data (i.e. text) is found.
3. An end tag is parsed (`</quote>`).

Each time such an event occurs the parser reports it so that a suitable event handling routine can be invoked.

Whilst this is a fast and simple approach to read XML documents manipulation is difficult because data are not stored, simply handled and discarded serially. This is when the DOM interface comes handy.

The Qt XML module provides an abstract class, QXmlReader, that defines the interface for potential SAX2 readers. At the moment Qt ships with one reader implementation, QXmlSimpleReader.

The reader reports parsing events through special handler classes. In Qt the following ones are available:

- QXmlContentHandler reports events related to the content of a document (e.g. the start tag or characters).
- QXmlDTDHandler reports events related to the DTD (e.g. notation declarations).
- QXmlErrorHandler reports errors or warnings that occurred during parsing.
- QXmlEntityResolver reports external entities during parsing and allows the user to resolve external entities him- or herself instead of leaving it to the reader.

- QXmlDeclHandler reports further DTD related events (e.g. attribute declarations). Usually users are not interested in them, but under certain circumstances this class comes handy.
- QXmlLexicalHandler reports events related to the lexical structure of the document (the beginning of the DTD, comments etc.). Occasionally this might be useful.

These classes are abstract classes describing the interface. The QXmlDefaultHandler class provides a "do nothing" default implementation for all of them. Therefore users need to overload only the QXmlDefaultHandler functions they are interested in.

To read input XML data a special class QXmlInputSource is used.

Apart from the already mentioned ones the following SAX2 support classes provide the user with useful functionality:

- QXmlAttributes is used to pass attributes in a start element event.
- QXmlLocator is used to obtain the actual parsing position of an event.
- QXmlNamespaceSupport is used to easily implement namespace support for a reader. Note that namespaces do not change the parsing behavior. They are only reported through the handler.

Features

The behaviour of an XML reader depends on whether it supports certain optional features or not. As an example a reader can have the feature "report attributes used for namespace declarations and prefixes along with the local name of a tag". Like every other feature this has a unique name represented by a URI: it is called *http://xml.org/sax/features/namespace-prefixes*.

The Qt SAX2 implementation allows you to find out whether the reader has this ability using QXmlReader::hasFeature(). If the return value is TRUE it is possible to turn the relevant feature on and off. To do this use QXmlReader::setFeature(). Whether a supported feature is on or off (TRUE or FALSE) can be queried using QXmlReader::feature().

Consider the example

A reader not supporting the *http://xml.org/sax/features/namespace-prefixes* feature would clearly report the element name *document* but not its attributes *xmlns:book* and *xmlns* with their values. A reader with the feature *http://xml.org/sax/features/namespace-prefixes* reports the namespace attributes if QXmlReader::feature() is TRUE and disregards them if the feature is FALSE.

Other features include *http://xml.org/sax/features/namespace* (namespace processing, implies *http://xml.org/sax/features/namespace-prefixes*) or *http://xml.org/sax/features/validation* (the ability to report validation errors).

Whilst SAX2 leaves it to the user to define and implement whatever features are required, support for *http://xml.org/sax/features/namespace* (and thus *http://xml.org/sax/features/namespace-prefixes*) is mandantory. Accordingly QXmlSimpleReader, the implementation of QXmlReader that comes with the Qt XML module, supports both of them, and therefore can do namespace processing.

Being a non-validating parser QXmlSimpleReader does not support *http://xml.org/sax/features/validation* and other features.

Namespace support via features

As we have seen in the previous section we can configure the behavior of the reader when it comes to namespace processing. This is done by setting and unsetting the *http://xml.org/sax/features/namespaces* and *http://xml.org/sax/features/namespace-prefixes* features.

They influence the reporting behavior in the following way:

1. Namespace prefixes and local parts of elements and attributes can be reported.
2. The qualified names of elements and attributes are reported.
3. QXmlContentHandler::startPrefixMapping() and QXmlContentHandler::endPrefixMapping() are called by the reader.
4. Attributes that declare namespaces (i.e. the attribute *xmlns* and attributes starting with *xmlns*:) are reported.

Consider the following element:

With *http://xml.org/sax/features/namespace-prefixes* set to TRUE the reader will report four attributes, with the *namespace-prefixes* feature set to FALSE only three: The *xmlns:fnord* attribute defining a namespace is then "unvisible" for the reader.

The *http://xml.org/sax/features/namespaces* feature on the other hand is responsible for reporting local names, namespace prefixes and -URIs. With *http://xml.org/sax/features/namespaces* set to TRUE the parser will report *title* as the local name of *fnord:title* attribute, *fnord* being the namespace prefix and *http://trolltech.com/fnord/* as the namespace URI. When *http://xml.org/sax/features/namespaces* is FALSE none of them are reported.

In the current implementation the Qt XML classes follow the definition that the prefix *xmlns* itself isn't associated with any namespace at all (see http://www.w3.org/TR/1999/REC-xml-names-19990114/#ns-using). Therefore even with *http://xml.org/sax/features/namespaces* and *http://xml.org/sax/features/namespace-prefixes* both set to TRUE the reader won't return either a local name, a namespace prefix or a namespace URI for *xmlns:fnord*.

This might be changed in the future following the W3C suggestion http://www.w3.org/2000/xmlns/ to associate *xmlns* with the namespace *http://www.w3.org/2000/xmlns*.

As the SAX2 standard suggests QXmlSimpleReader by default has *http://xml.org/sax/features/namespaces* set to TRUE and *http://xml.org/sax/features/namespace-prefixes* set to FALSE. When changing this behavior using QXmlSimpleReader::setFeature() note that the combination of both features set to FALSE is illegal.

For a practical demonstration of how the two features affect the output of the reader run the tagreader with features example.

Summary

QXmlSimpleReader implements the following behavior:

(namespaces, namespace-prefixes)	Namespace prefix and local part	Qualified names	Prefix mapping	xmlns attributes
(TRUE, FALSE)	Yes	Yes*	Yes	No
(TRUE, TRUE)	Yes	Yes	Yes	Yes
(FALSE, TRUE)	No*	Yes	No*	Yes
(FALSE, FALSE)	Illegal			

For the entries marked with a "*", SAX does not require a particuliar behavior.

Properties

Properties are a more general concept. They also have a unique name, represented as an URI, but their value is `void*`. Thus nearly everything can be used as a property value. This concept involves some danger, though: there are no means to ensure type-safety; the user must take care that he or she passes the correct type. Properties are useful if a reader supports special handler classes.

The URIs used for features and properties often look like URLs, e.g. http://xml.org/sax/features/namespace. This does not mean that whatsoever data is required at this address. It is simply a way to define unique names.

Everybody can define and use new SAX2 properties for his or her readers. Property support is however not required.

To set or query properties the following functions are provided: QXmlReader::setProperty(), QXmlReader::property() and QXmlReader::hasProperty().

Further reading

For a practical example on how to use the Qt SAX2 classes see the tagreader walkthrough.

More information about XML (e.g. namespaces) can be found in the introduction to the Qt XML module.

The Qt DOM classes

Introduction to DOM

DOM provides an interface to access and change the content and structure of an XML file. It makes a hierarchical view of the document (tree) available with the root element of the XML file serving as its root. Thus — in contrast to the SAX2 interface — an object model of the document is resident in memory after parsing which makes manipulation easy.

In the Qt implementation of the DOM all nodes in the document tree are subclasses of QDomNode. The document itself is represented as a QDomDocument object.

Here are the available node classes and their potential children classes:

- QDomDocument: Possible children are
  - QDomElement (at most one)
  - QDomProcessingInstruction
  - QDomComment
  - QDomDocumentType
- QDomDocumentFragment: Possible children are
  - QDomElement
  - QDomProcessingInstruction
  - QDomComment
  - QDomText
  - QDomCDATASection
  - QDomEntityReference
- QDomDocumentType: No children
- QDomEntityReference: Possible children are
  - QDomElement
  - QDomProcessingInstruction
  - QDomComment
  - QDomText
  - QDomCDATASection
  - QDomEntityReference
- QDomElement: Possible children are

  - QDomElement
  - QDomText
  - QDomComment
  - QDomProcessingInstruction
  - QDomCDATASection
  - QDomEntityReference

- QDomAttr: Possible children are
  - QDomText
  - QDomEntityReference
- QDomProcessingInstruction: No children
- QDomComment: No children
- QDomText: No children
- QDomCDATASection: No children
- QDomEntity: Possible children are
  - QDomElement
  - QDomProcessingInstruction
  - QDomComment
  - QDomText
  - QDomCDATASection
  - QDomEntityReference
- QDomNotation: No children

With QDomNodeList and QDomNamedNodeMap two collection classes are provided: QDomNodeList is a list of nodes whereas QDomNamedNodeMap is used to handle unordered sets of nodes (often used for attributes).

The QDomImplementation class allows the user to query features of the DOM implementation.

Further reading

To get started please refer to the QDomDocument documentation that describes basic usage.

More information about Qt and XML can be found in the Introduction to the Qt XML module.

An introduction to namespaces

Parts of the Qt XML module documentation assume that you are familiar with XML namespaces. Here we present a brief introduction; skip to Qt XML documentation conventions if you know this material.

Namespaces are a concept introduced into XML to allow a more modular design. With their help data processing software can easily resolve naming conflicts in XML documents.

Consider the following example:

```


  Practical XML


    A Namespace Called fnord

```

Here we find three different uses of the name *title*. If you wish to process this document you will encounter problems because each of the *titles* should be displayed in a different manner — even though they have the same name.

The solution would be to have some means of identifying the first occurrence of *title* as the title of a book, i.e. to use the *title* element of a book namespace to distinguish it from for example the chapter title, e.g.:

```
Practical XML
```

book in this case is a *prefix* denoting the namespace.

Before we can apply a namespace to element or attribute names we must declare it.

Namespaces are URIs like *http://trolltech.com/fnord/book/*. This does not mean that data must be available at this address; the URI is simply used to provide a unique name.

We declare namespaces in the same way as attributes; strictly speaking they *are* attributes. To make for example *http://trolltech.com/fnord/* the document's default XML namespace *xmlns* we write

```
xmlns="http://trolltech.com/fnord/"
```

To distinguish the *http://trolltech.com/fnord/book/* namespace from the default, we have to supply it with a prefix:

```
xmlns:book="http://trolltech.com/fnord/book/"
```

A namespace that is declared like this can be applied to element and attribute names by prepending the appropriate prefix and a ":" delimiter. We have already seen this with the *book:title* element.

Element names without a prefix belong to the default namespace. This rule does not apply to attributes: an attribute without a prefix does not belong to any of the declared XML namespaces at all. Attributes always belong to the "traditional" namespace of the element in which they appear. A "traditional" namespace is not an XML namespace, it simply means that all attribute names belonging to one element must be different. Later we will see how to assign an XML namespace to an attribute.

Due to the fact that attributes without prefixes are not in any XML namespace there is no collision between the attribute *title* (that belongs to the *author* element) and for example the *title* element within a *chapter*.

Let's clarify matters with an example:

```


  Practical XML


    A Namespace Called fnord


```

Within the *document* element we have two namespaces declared. The default namespace *http://trolltech.com/fnord/* applies to the *book* element, the *chapter* element, the appropriate *title* element and of course to *document* itself.

The *book:author* and *book:title* elements belong to the namespace with the URI *http://trolltech.com/fnord/book/*.

The two *book:author* attributes *title* and *name* have no XML namespace assigned. They are only members of the "traditional" namespace of the element *book:author*, meaning that for example two *title* attributes in *book:author* are forbidden.

In the above example we circumvent the last rule by adding a *title* attribute from the *http://trolltech.com/fnord/* namespace to *book:author*: the *fnord:title* comes from the namespace with the prefix *fnord* that is declared in the *book:author* element.

Clearly the *fnord* namespace has the same namespace URI as the default namespace. So why didn't we simply use the default namespace we'd already declared? The answer is quite complex:

- attributes without a prefix don't belong to any XML namespace at all, even not to the default namespace;
- additionally omitting the prefix would lead to a *title-title* clash;
- writing it as *xmlns:title* would declare a new namespace with the prefix *title* instead of applying the default *xmlns* namespace.

With the Qt XML classes elements and attributes can be accessed in two ways: either by refering to their qualified names consisting of the namespace prefix and the "real" name (or *local* name) or by the combination of local name and namespace URI.

More information on XML namespaces can be found at http://www.w3.org/TR/REC-xml-names/.

Conventions used in Qt XML documentation

The following terms are used to distinguish the parts of names within the context of namespaces:

- The *qualified name* is the name as it appears in the document. (In the above example *book:title* is a qualified name.)
- A *namespace prefix* in a qualified name is the part to the left of the ":". (*book* is the namespace prefix in *book:title*.)
- The *local part* of a name (also refered to as the *local name*) appears to the right of the ":". (Thus *title* is the local part of *book:title*.)
- The *namespace URI* ("Uniform Resource Identifier") is a unique identifier for a namespace. It looks like a URL (e.g. *http://trolltech.com/fnord/*) but does not require data to be accessible by the given protocol at the named address.

Elements without a ":" (like *chapter* in the example) do not have a namespace prefix. In this case the local part and the qualified name are identical (i.e. *chapter*).

QDomAttr Class Reference

The QDomAttr class represents one attribute of a QDomElement.

This class is part of the **XML** module.

`#include <qdom.h>`

Inherits QDomNode [p. 53].

Public Members

- **QDomAttr** ()
- **QDomAttr** (const QDomAttr & x)
- QDomAttr & **operator=** (const QDomAttr & x)
- **~QDomAttr** ()
- virtual QString **name** () const
- virtual bool **specified** () const
- virtual QDomElement **ownerElement** () const
- virtual QString **value** () const
- virtual void **setValue** (const QString & v)
- virtual QDomNode::NodeType **nodeType** () const
- virtual bool **isAttr** () const

Detailed Description

The QDomAttr class represents one attribute of a QDomElement.

For example, the following piece of XML gives an element with no children, but two attributes:

One can use the attributes of an element with code like this:

```
QDomElement e = //...
//...
QDomAttr a = e.attributeNode( "href" );
cout << a.value() << endl // gives "http://www.trolltech.com"
a.setValue( "http://doc.trolltech.com" ); // change the node's attribute
QDomAttr a2 = e.attributeNode( "href" );
cout << a2.value() << endl // gives "http://doc.trolltech.com"
```

This example also shows that changing an attribute received from an element changes the attribute of the element. If you do not want to change the value of the element's attribute you have to use cloneNode() to get an independent copy of the attribute.

QDomAttr can return the name() and value() of an attribute. An attribute's value is set with setValue(). If specified returns TRUE the value was either set in the document or set with setValue(); otherwise the value hasn't been set. The node this attribute is attached to (if any) is returned by ownerElement().

For further information about the Document Object Model see http://www.w3.org/TR/REC-DOM-Level-1/ and http://www.w3.org/TR/DOM-Level-2-Core/. For a more general introduction of the DOM implementation see the QDomDocument documentation.

See also XML.

Member Function Documentation

QDomAttr::QDomAttr ()

Constructs an empty attribute.

QDomAttr::QDomAttr (const QDomAttr & x)

Constructs a copy of *x*.

The data of the copy is shared (shallow copy): modifying one node will also change the other. If you want to make a deep copy, use cloneNode().

QDomAttr::~QDomAttr ()

Destroys the object and frees its resources.

bool QDomAttr::isAttr () const [virtual]

Returns TRUE.

Reimplemented from QDomNode [p. 58].

QString QDomAttr::name () const [virtual]

Returns the name of the attribute.

QDomNode::NodeType QDomAttr::nodeType () const [virtual]

Returns AttributeNode.

Reimplemented from QDomNode [p. 62].

QDomAttr & QDomAttr::operator= (const QDomAttr & x)

Assigns *x* to this DOM attribute.

The data of the copy is shared (shallow copy): modifying one node will also change the other. If you want to make a deep copy, use cloneNode().

QDomElement QDomAttr::ownerElement () const [virtual]

Returns the element node, this attribute is attached to. If this attribute is not attached to any element, a null element node is returned (i.e. a node for which QDomNode::isNull() returns TRUE).

void QDomAttr::setValue (const QString & v) [virtual]

Sets the value of the attribute to *v*.

See also value() [p. 14].

bool QDomAttr::specified () const [virtual]

Returns TRUE if the attribute has either been expicitly specified in the XML document or was set by the user with setValue(). Returns FALSE if the value hasn't been specified or set.

See also setValue() [p. 14].

QString QDomAttr::value () const [virtual]

Returns the value of the attribute. Returns a null string if the attribute has not been specified.

See also specified() [p. 14] and setValue() [p. 14].

QDomCDATASection Class Reference

The QDomCDATASection class represents an XML CDATA section.

This class is part of the **XML** module.

`#include <qdom.h>`

Inherits QDomText [p. 75].

Public Members

- **QDomCDATASection** ()
- **QDomCDATASection** (const QDomCDATASection & x)
- QDomCDATASection & **operator=** (const QDomCDATASection & x)
- **~QDomCDATASection** ()
- virtual QDomNode::NodeType **nodeType** () const
- virtual bool **isCDATASection** () const

Detailed Description

The QDomCDATASection class represents an XML CDATA section.

CDATA sections are used to escape blocks of text containing characters that would otherwise be regarded as markup. The only delimiter that is recognized in a CDATA section is the "]]>" string that terminates the CDATA section. CDATA sections cannot be nested. Their primary purpose is for including material such as XML fragments, without needing to escape all the delimiters.

Adjacent QDomCDATASection nodes are not merged by the QDomNode::normalize() function.

For further information about the Document Object Model see http://www.w3.org/TR/REC-DOM-Level-1/ and http://www.w3.org/TR/DOM-Level-2-Core/. For a more general introduction of the DOM implementation see the QDomDocument documentation.

See also XML.

Member Function Documentation

QDomCDATASection::QDomCDATASection ()

Constructs an empty CDATA section. To create a CDATA section with content, use the QDomDocument::createCDATASection() function.

QDomCDATASection::QDomCDATASection (const QDomCDATASection & x)

Constructs a copy of *x*.

The data of the copy is shared (shallow copy): modifying one node will also change the other. If you want to make a deep copy, use cloneNode().

QDomCDATASection::~QDomCDATASection ()

Destroys the object and frees its resources.

bool QDomCDATASection::isCDATASection () const [virtual]

Returns TRUE.

Reimplemented from QDomNode [p. 58].

QDomNode::NodeType QDomCDATASection::nodeType () const [virtual]

Returns `CDATASection`.

Reimplemented from QDomText [p. 76].

QDomCDATASection & QDomCDATASection::operator= (const QDomCDATASection & x)

Assigns *x* to this CDATA section.

The data of the copy is shared (shallow copy): modifying one node will also change the other. If you want to make a deep copy, use cloneNode().

QDomCharacterData Class Reference

The QDomCharacterData class represents a generic string in the DOM.

This class is part of the **XML** module.

`#include <qdom.h>`

Inherits QDomNode [p. 53].

Inherited by QDomText [p. 75] and QDomComment [p. 20].

Public Members

- **QDomCharacterData** ()
- **QDomCharacterData** (const QDomCharacterData & x)
- QDomCharacterData & **operator=** (const QDomCharacterData & x)
- **~QDomCharacterData** ()
- virtual QString **substringData** (unsigned long offset, unsigned long count)
- virtual void **appendData** (const QString & arg)
- virtual void **insertData** (unsigned long offset, const QString & arg)
- virtual void **deleteData** (unsigned long offset, unsigned long count)
- virtual void **replaceData** (unsigned long offset, unsigned long count, const QString & arg)
- virtual uint **length** () const
- virtual QString **data** () const
- virtual void **setData** (const QString & v)
- virtual QDomNode::NodeType **nodeType** () const
- virtual bool **isCharacterData** () const

Detailed Description

The QDomCharacterData class represents a generic string in the DOM.

Character data as used in XML specifies a generic data string. More specialized versions of this class are QDomText, QDomComment and QDomCDATASection.

The data string is set with setData() and retrieved with data(). You can retrieve a portion of the data string using substringData(). Extra data can be appended with appendData(), or inserted with insertData(). Portions of the data string can be deleted with deleteData() or replaced with replaceData(). The length of the data string is returned by length().

The node type of the node containing this character data is returned by nodeType().

See also QDomText [p. 75], QDomComment [p. 20], QDomCDATASection [p. 15] and XML.

Member Function Documentation

QDomCharacterData::QDomCharacterData ()

Constructs an empty character data object.

QDomCharacterData::QDomCharacterData (const QDomCharacterData & x)

Constructs a copy of *x*.

The data of the copy is shared (shallow copy): modifying one node will also change the other. If you want to make a deep copy, use cloneNode().

QDomCharacterData::~QDomCharacterData ()

Destroys the object and frees its resources.

void QDomCharacterData::appendData (const QString & arg) [virtual]

Appends the string *arg*, to the stored string.

QString QDomCharacterData::data () const [virtual]

Returns the string stored in this object.

If the node is a null node, it will return a null string.

void QDomCharacterData::deleteData (unsigned long offset, unsigned long count) [virtual]

Deletes a substring of length *count* from position *offset*.

void QDomCharacterData::insertData (unsigned long offset, const QString & arg) [virtual]

Inserts the string *arg* into the stored string at position *offset*.

bool QDomCharacterData::isCharacterData () const [virtual]

Returns TRUE.

Reimplemented from QDomNode [p. 59].

uint QDomCharacterData::length () const [virtual]

Returns the length of the stored string.

QDomNode::NodeType QDomCharacterData::nodeType () const [virtual]

Returns the type of node this object refers to (i.e. TextNode, CDATASectionNode, CommentNode or CharacterDataNode). For a null node CharacterDataNode is returned.

Reimplemented from QDomNode [p. 62].

Reimplemented in QDomText and QDomComment.

QDomCharacterData & QDomCharacterData::operator= (const QDomCharacterData & x)

Assigns *x* to this character data.

The data of the copy is shared (shallow copy): modifying one node will also change the other. If you want to make a deep copy, use cloneNode().

void QDomCharacterData::replaceData (unsigned long offset, unsigned long count, const QString & arg) [virtual]

Replaces the substring of length *count* starting at position *offset* with the string *arg*.

void QDomCharacterData::setData (const QString & v) [virtual]

Sets the string of this object to *v*.

QString QDomCharacterData::substringData (unsigned long offset, unsigned long count) [virtual]

Returns the substring of length *count* from position *offset*.

QDomComment Class Reference

The QDomComment class represents an XML comment.

This class is part of the **XML** module.

`#include <qdom.h>`

Inherits QDomCharacterData [p. 17].

Public Members

- **QDomComment** ()
- **QDomComment** (const QDomComment & x)
- QDomComment & **operator=** (const QDomComment & x)
- **~QDomComment** ()
- virtual QDomNode::NodeType **nodeType** () const
- virtual bool **isComment** () const

Detailed Description

The QDomComment class represents an XML comment.

A comment in the parsed XML such as this:

is represented by QDomComment objects in the parsed Dom tree.

For further information about the Document Object Model see http://www.w3.org/TR/REC-DOM-Level-1/ and http://www.w3.org/TR/DOM-Level-2-Core/. For a more general introduction of the DOM implementation see the QDomDocument documentation.

See also XML.

Member Function Documentation

QDomComment::QDomComment ()

Constructs an empty comment. To construct a comment with content, use the QDomDocument::createComment() function.

QDomComment::QDomComment (const QDomComment & x)

Constructs a copy of *x*.

The data of the copy is shared (shallow copy): modifying one node will also change the other. If you want to make a deep copy, use cloneNode().

QDomComment::~QDomComment ()

Destroys the object and frees its resources.

bool QDomComment::isComment () const [virtual]

Returns TRUE.

Reimplemented from QDomNode [p. 59].

QDomNode::NodeType QDomComment::nodeType () const [virtual]

Returns CommentNode.

Reimplemented from QDomCharacterData [p. 19].

QDomComment & QDomComment::operator= (const QDomComment & x)

Assigns *x* to this DOM comment.

The data of the copy is shared (shallow copy): modifying one node will also change the other. If you want to make a deep copy, use cloneNode().

QDomDocument Class Reference

The QDomDocument class represents an XML document.

This class is part of the **XML** module.

`#include <qdom.h>`

Inherits QDomNode [p. 53].

Public Members

- **QDomDocument** ()
- **QDomDocument** (const QString & name)
- **QDomDocument** (const QDomDocumentType & doctype)
- **QDomDocument** (const QDomDocument & x)
- QDomDocument & **operator=** (const QDomDocument & x)
- **~QDomDocument** ()
- QDomElement **createElement** (const QString & tagName)
- QDomDocumentFragment **createDocumentFragment** ()
- QDomText **createTextNode** (const QString & value)
- QDomComment **createComment** (const QString & value)
- QDomCDATASection **createCDATASection** (const QString & value)
- QDomProcessingInstruction **createProcessingInstruction** (const QString & target, const QString & data)
- QDomAttr **createAttribute** (const QString & name)
- QDomEntityReference **createEntityReference** (const QString & name)
- QDomNodeList **elementsByTagName** (const QString & tagname) const
- QDomNode **importNode** (const QDomNode & importedNode, bool deep)
- QDomElement **createElementNS** (const QString & nsURI, const QString & qName)
- QDomAttr **createAttributeNS** (const QString & nsURI, const QString & qName)
- QDomNodeList **elementsByTagNameNS** (const QString & nsURI, const QString & localName)
- QDomElement **elementById** (const QString & elementId)
- QDomDocumentType **doctype** () const
- QDomImplementation **implementation** () const
- QDomElement **documentElement** () const
- bool **setContent** (const QCString & buffer, bool namespaceProcessing, QString * errorMsg = 0, int * errorLine = 0, int * errorColumn = 0)
- bool **setContent** (const QByteArray & buffer, bool namespaceProcessing, QString * errorMsg = 0, int * errorLine = 0, int * errorColumn = 0)
- bool **setContent** (const QString & text, bool namespaceProcessing, QString * errorMsg = 0, int * errorLine = 0, int * errorColumn = 0)
- bool **setContent** (QIODevice * dev, bool namespaceProcessing, QString * errorMsg = 0, int * errorLine = 0, int * errorColumn = 0)

- bool **setContent** (const QCString & buffer, QString * errorMsg = 0, int * errorLine = 0, int * errorColumn = 0)
- bool **setContent** (const QByteArray & buffer, QString * errorMsg = 0, int * errorLine = 0, int * errorColumn = 0)
- bool **setContent** (const QString & text, QString * errorMsg = 0, int * errorLine = 0, int * errorColumn = 0)
- bool **setContent** (QIODevice * dev, QString * errorMsg = 0, int * errorLine = 0, int * errorColumn = 0)
- virtual QDomNode::NodeType **nodeType** () const
- virtual bool **isDocument** () const
- QString **toString** () const
- QCString **toCString** () const

Detailed Description

The QDomDocument class represents an XML document.

The QDomDocument class represents the entire XML document. Conceptually, it is the root of the document tree, and provides the primary access to the document's data.

Since elements, text nodes, comments, processing instructions, etc. cannot exist outside the context of a document, the document class also contains the factory functions needed to create these objects. The node objects created have an ownerDocument() function which associates them with the document within whose context they were created. The DOM classes that will be used most often are QDomNode, QDomDocument, QDomElement and QDomText.

The parsed XML is represented internally by a tree of objects that can be accessed using the various QDom classes. All QDom classes only *reference* objects in the internal tree. The internal objects in the DOM tree will get deleted, once the last QDom object referencing them and the QDomDocument itself are deleted.

Creation of elements, text nodes, etc. is done via the various factory functions provided in this class. Using the default constructors of the QDom classes will only result in empty objects, that cannot be manipulated or inserted into the Document.

The QDomDocument class has several functions for creating document data, for example, createElement(), createTextNode(), createComment(), createCDATASection(), createProcessingInstruction(), createAttribute() and createEntityReference(). Some of these functions have versions that support namespaces, i.e. createElementNS() and createAttributeNS(). The createDocumentFragment() function is used to hold parts of the document, e.g. for complex documents.

The entire content of the document is set with setContent(). This function parses the string it is passed as an XML document and creates the DOM tree that represents the document. The root element is available using documentElement(). The textual representation of the document can be obtained using toString().

It is possible to insert a node from another document into the document using importNode().

You can obtain a list of all the elements that have a particular tag with elementsByTagName() or with elementsByTagNameNS().

The QDom classes are typically used as follows:

```
QDomDocument doc( "mydocument" );
QFile f( "mydocument.xml" );
if ( !f.open( IO_ReadOnly ) )
    return;
if ( !doc.setContent( &f ) ) {
    f.close();
    return;
}
f.close();

// print out the element names of all elements that are direct children
// of the outermost element.
QDomElement docElem = doc.documentElement();

QDomNode n = docElem.firstChild();
while( !n.isNull() ) {
    QDomElement e = n.toElement(); // try to convert the node to an element.
    if( !e.isNull() ) {
        cout << e.tagName() << endl; // the node really is an element.
    }
    n = n.nextSibling();
}

// Here we append a new element to the end of the document
QDomElement elem = doc.createElement( "img" );
elem.setAttribute( "src", "myimage.png" );
docElem.appendChild( elem );
```

Once `doc` and `elem` go out of scode, the whole internal tree representing the XML document will get deleted.

To create a document using DOM use code like this:

```
QDomDocument doc( "MyML" );
QDomElement root = doc.createElement( "MyML" );
doc.appendChild( root );

QDomElement tag = doc.createElement( "Greeting" );
root.appendChild( tag );

QDomText t = doc.createTextNode( "Hello World" );
tag.appendChild( t );

QString xml = doc.toString();
```

For further information about the Document Object Model see http://www.w3.org/TR/REC-DOM-Level-1/ and http://www.w3.org/TR/DOM-Level-2-Core/. For a more general introduction of the DOM implementation see the QDomDocument documentation.

See also XML.

Member Function Documentation

QDomDocument::QDomDocument ()

Constructs an empty document.

QDomDocument::QDomDocument (const QString & name)

Creates a document and sets the name of the document type to *name*.

QDomDocument::QDomDocument (const QDomDocumentType & doctype)

Creates a document with the document type *doctype*.

See also QDomImplementation::createDocumentType() [p. 47].

QDomDocument::QDomDocument (const QDomDocument & x)

Constructs a copy of *x*.

The data of the copy is shared (shallow copy): modifying one node will also change the other. If you want to make a deep copy, use cloneNode().

QDomDocument::~QDomDocument ()

Destroys the object and frees its resources.

QDomAttr QDomDocument::createAttribute (const QString & name)

Creates a new attribute called *name* that can be inserted into an element, e.g. using QDomElement::setAttributeNode().

See also createAttributeNS() [p. 25].

QDomAttr QDomDocument::createAttributeNS (const QString & nsURI, const QString & qName)

Creates a new attribute with namespace support that can be inserted into an element. The name of the attribute is *qName* and the namespace URI is *nsURI*. This function also sets QDomNode::prefix() and QDomNode::localName() to appropriate values (depending on *qName*).

See also createAttribute() [p. 25].

QDomCDATASection QDomDocument::createCDATASection (const QString & value)

Creates a new CDATA section for the string *value* that can be inserted into the document, e.g. using QDomNode::appendChild().

See also QDomNode::appendChild() [p. 56], QDomNode::insertBefore() [p. 58] and QDomNode::insertAfter() [p. 58].

QDomComment QDomDocument::createComment (const QString & value)

Creates a new comment for the string *value* that can be inserted into the document, e.g. using QDomNode::appendChild().

See also QDomNode::appendChild() [p. 56], QDomNode::insertBefore() [p. 58] and QDomNode::insertAfter() [p. 58].

QDomDocumentFragment QDomDocument::createDocumentFragment ()

Creates a new document fragment, that can be used to hold parts of the document, e.g. when doing complex manipulations of the document tree.

QDomElement QDomDocument::createElement (const QString & tagName)

Creates a new element called *tagName* that can be inserted into the DOM tree, e.g. using QDomNode::appendChild().

See also createElementNS() [p. 26], QDomNode::appendChild() [p. 56], QDomNode::insertBefore() [p. 58] and QDomNode::insertAfter() [p. 58].

QDomElement QDomDocument::createElementNS (const QString & nsURI, const QString & qName)

Creates a new element with namespace support that can be inserted into the DOM tree. The name of the element is *qName* and the namespace URI is *nsURI*. This function also sets QDomNode::prefix() and QDomNode::localName() to appropriate values (depending on *qName*).

See also createElement() [p. 25].

QDomEntityReference QDomDocument::createEntityReference (const QString & name)

Creates a new entity reference called *name* that can be inserted into the document, e.g. using QDomNode::appendChild().

See also QDomNode::appendChild() [p. 56], QDomNode::insertBefore() [p. 58] and QDomNode::insertAfter() [p. 58].

QDomProcessingInstruction QDomDocument::createProcessingInstruction (const QString & target, const QString & data)

Creates a new processing instruction that can be inserted into the document, e.g. using QDomNode::appendChild(). This function sets the target for the processing instruction to *target* and the data to *data*.

See also QDomNode::appendChild() [p. 56], QDomNode::insertBefore() [p. 58] and QDomNode::insertAfter() [p. 58].

QDomText QDomDocument::createTextNode (const QString & value)

Creates a text node for the string *value* that can be inserted into the document tree, e.g. using QDomNode::appendChild().

See also QDomNode::appendChild() [p. 56], QDomNode::insertBefore() [p. 58] and QDomNode::insertAfter() [p. 58].

QDomDocumentType QDomDocument::doctype () const

Returns the document type of this document.

QDomElement QDomDocument::documentElement () const

Returns the root element of the document.

QDomElement QDomDocument::elementById (const QString & elementId)

Returns the element whose ID is equal to *elementId*. If no element with the ID was found, this function returns a null element.

Since the actual version of the QDomClasses does not know which attributes are element IDs, this function returns always a null element. This may change in a future version.

QDomNodeList QDomDocument::elementsByTagName (const QString & tagname) const

Returns a QDomNodeList, that contains all the elements in the document with the name *tagname*. The order of the node list is the order they are encountered in a preorder traversal of the element tree.

See also elementsByTagNameNS() [p. 27] and QDomElement::elementsByTagName() [p. 38].

QDomNodeList QDomDocument::elementsByTagNameNS (const QString & nsURI, const QString & localName)

Returns a QDomNodeList that contains all the elements in the document with the local name *localName* and a namespace URI of *nsURI*. The order of the node list, is the order they are encountered in a preorder traversal of the element tree.

See also elementsByTagName() [p. 27] and QDomElement::elementsByTagNameNS() [p. 39].

QDomImplementation QDomDocument::implementation () const

Returns a QDomImplementation object.

QDomNode QDomDocument::importNode (const QDomNode & importedNode, bool deep)

Imports the node *importedNode* from another document to this document. *importedNode* remains in the original document; this function creates a copy of it that can be used within this document.

This function returns the imported node that belongs to this document. The returned node has no parent. It is not possible to import QDomDocument and QDomDocumentType nodes. In those cases this function returns a null node.

If *deep* is TRUE, this function imports not only the node *importedNode* but the whole subtree; if it is FALSE, only the *importedNode* is imported. The argument *deep* has no effect on QDomAttr and QDomEntityReference nodes, since the descendants of QDomAttr nodes are always imported and those of QDomEntityReference nodes are never imported.

The behavior of this function is slightly different depending on the node types:

- QDomAttr - The owner element is set to 0 and the specified flag is set to TRUE on the generated attribute. The whole subtree of *importedNode* is always imported for attribute nodes - *deep* has no effect.
- QDomDocument - Document nodes cannot be imported.
- QDomDocumentFragment - If *deep* is TRUE, this function imports the whole document fragment, otherwise it only generates an empty document fragment.
- QDomDocumentType - Document type nodes cannot be imported.
- QDomElement - Attributes for which QDomAttr::specified() is TRUE are also imported, other attributes are not imported. If *deep* is TRUE, this function also imports the subtree of *importedNode*, otherwise it imports only the element node (and some attributes, see above).
- QDomEntity - Entity nodes can be imported, but at the moment there is no way to use them since the document type is readonly in DOM level 2.
- QDomEntityReference - Descendants of entity reference nodes are never imported - *deep* has no effect.

- QDomNotation - Notation nodes can be imported, but at the moment there is no way to use them since the document type is readonly in DOM level 2.
- QDomProcessingInstruction - The target and value of the processing instruction is copied to the new node.
- QDomText, QDomCDATASection and QDomComment - The text is copied to the new node.

See also QDomElement::setAttribute() [p. 40], QDomNode::insertBefore() [p. 58], QDomNode::insertAfter() [p. 58], QDomNode::replaceChild() [p. 64], QDomNode::removeChild() [p. 64] and QDomNode::appendChild() [p. 56].

bool QDomDocument::isDocument () const [virtual]

Returns TRUE.

Reimplemented from QDomNode [p. 59].

QDomNode::NodeType QDomDocument::nodeType () const [virtual]

Returns DocumentNode.

Reimplemented from QDomNode [p. 62].

QDomDocument & QDomDocument::operator= (const QDomDocument & x)

Assigns *x* to this DOM document.

The data of the copy is shared (shallow copy): modifying one node will also change the other. If you want to make a deep copy, use cloneNode().

bool QDomDocument::setContent (const QString & text, bool namespaceProcessing, QString * errorMsg = 0, int * errorLine = 0, int * errorColumn = 0)

This function parses the string *text* and sets it as the content of the document. If *namespaceProcessing* is TRUE, the parser recognizes namespaces in the XML file and sets the prefix name, local name and namespace URI to appropriate values. If *namespaceProcessing* is FALSE, the parser does no namespace processing when it reads the XML file.

If a parse error occurs, the function returns FALSE; otherwise TRUE. If a parse error occurs the error message is placed in **errorMsg*, the line number in **errorLine* and the column number in **errorColumn*. These error variables will only be populated if they are non-null.

If *namespaceProcessing* is TRUE, the function QDomNode::prefix() returns a string for all elements and attributes. It returns an empty string if the element or attribute has no prefix.

If *namespaceProcessing* is FALSE, the functions QDomNode::prefix(), QDomNode::localName() and QDomNode::namespaceURI() return a null string.

See also QDomNode::namespaceURI() [p. 61], QDomNode::localName() [p. 61], QDomNode::prefix() [p. 64], QString::isNull() [Datastructures and String Handling with Qt] and QString::isEmpty() [Datastructures and String Handling with Qt].

bool QDomDocument::setContent (const QCString & buffer, bool namespaceProcessing, QString * errorMsg = 0, int * errorLine = 0, int * errorColumn = 0)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

This function reads the XML document from the C string *buffer*.

bool QDomDocument::setContent (const QByteArray & buffer, bool namespaceProcessing, QString * errorMsg = 0, int * errorLine = 0, int * errorColumn = 0)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

This function reads the XML document from the byte array *buffer*.

bool QDomDocument::setContent (QIODevice * dev, bool namespaceProcessing, QString * errorMsg = 0, int * errorLine = 0, int * errorColumn = 0)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

This function reads the XML document from the IO device *dev*.

bool QDomDocument::setContent (const QCString & buffer, QString * errorMsg = 0, int * errorLine = 0, int * errorColumn = 0)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

This function reads the XML document from the C string *buffer*.

No namespace processing is done.

bool QDomDocument::setContent (const QByteArray & buffer, QString * errorMsg = 0, int * errorLine = 0, int * errorColumn = 0)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

This function reads the XML document from the byte array *buffer*.

No namespace processing is done.

bool QDomDocument::setContent (const QString & text, QString * errorMsg = 0, int * errorLine = 0, int * errorColumn = 0)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

This function reads the XML document from the string *text*.

No namespace processing is done.

bool QDomDocument::setContent (QIODevice * dev, QString * errorMsg = 0, int * errorLine = 0, int * errorColumn = 0)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

This function reads the XML document from the IO device *dev*.

No namespace processing is done.

QCString QDomDocument::toCString () const

Converts the parsed document back to its textual representation and returns a QCString for that is encoded in UTF-8.

See also toString() [p. 30].

QString QDomDocument::toString () const

Converts the parsed document back to its textual representation.

See also toCString() [p. 30].

QDomDocumentFragment Class Reference

The QDomDocumentFragment class is a tree of QDomNodes which is not usually a complete QDomDocument.

This class is part of the **XML** module.

`#include <qdom.h>`

Inherits QDomNode [p. 53].

Public Members

- **QDomDocumentFragment** ()
- **QDomDocumentFragment** (const QDomDocumentFragment & x)
- QDomDocumentFragment & **operator=** (const QDomDocumentFragment & x)
- **~QDomDocumentFragment** ()
- virtual QDomNode::NodeType **nodeType** () const
- virtual bool **isDocumentFragment** () const

Detailed Description

The QDomDocumentFragment class is a tree of QDomNodes which is not usually a complete QDomDocument.

If you want to do complex tree operations it is useful to have a lightweight class to store nodes and their relations. QDomDocumentFragment stores a subtree of a document which does not necessarily represent a well-formed XML document.

QDomDocumentFragment is also useful if you want to group several nodes in a list and insert them all together as children of some node. In these cases QDomDocumentFragment can be used as a temporary container for this list of children.

The most important feature of QDomDocumentFragment is that it is treated in a special way by QDomNode::insertAfter(), QDomNode::insertBefore() and QDomNode::replaceChild(): instead of inserting the fragment itself, all the fragment's children are inserted.

See also XML.

Member Function Documentation

QDomDocumentFragment::QDomDocumentFragment ()

Constructs an empty document fragment.

QDomDocumentFragment::QDomDocumentFragment (const QDomDocumentFragment & x)

Constructs a copy of *x*.

The data of the copy is shared (shallow copy): modifying one node will also change the other. If you want to make a deep copy, use cloneNode().

QDomDocumentFragment::~QDomDocumentFragment ()

Destroys the object and frees its resources.

bool QDomDocumentFragment::isDocumentFragment () const [virtual]

This function reimplements QDomNode::isDocumentFragment().

See also nodeType() [p. 32] and QDomNode::toDocumentFragment() [p. 65].

Reimplemented from QDomNode [p. 59].

QDomNode::NodeType QDomDocumentFragment::nodeType () const [virtual]

Returns `DocumentFragment`.

See also isDocumentFragment() [p. 32] and QDomNode::toDocumentFragment() [p. 65].

Reimplemented from QDomNode [p. 62].

QDomDocumentFragment & QDomDocumentFragment::operator= (const QDomDocumentFragment & x)

Assigns *x* to this DOM document fragment.

The data of the copy is shared (shallow copy): modifying one node will also change the other. If you want to make a deep copy, use cloneNode().

QDomDocumentType Class Reference

The QDomDocumentType class is the representation of the DTD in the document tree.

This class is part of the **XML** module.

`#include <qdom.h>`

Inherits QDomNode [p. 53].

Public Members

- **QDomDocumentType** ()
- **QDomDocumentType** (const QDomDocumentType & n)
- QDomDocumentType & **operator=** (const QDomDocumentType & n)
- **~QDomDocumentType** ()
- virtual QString **name** () const
- virtual QDomNamedNodeMap **entities** () const
- virtual QDomNamedNodeMap **notations** () const
- virtual QString **publicId** () const
- virtual QString **systemId** () const
- virtual QString **internalSubset** () const
- virtual QDomNode::NodeType **nodeType** () const
- virtual bool **isDocumentType** () const

Detailed Description

The QDomDocumentType class is the representation of the DTD in the document tree.

The QDomDocumentType class allows read-only access to some of the data structures in the DTD: it can return a map of all entities() and notations(). In addition the function name() returns the name of the document type as specified in the <!DOCTYPE name> tag. This class also provides the publicId(), systemId() and internalSubset() functions.

See also QDomDocument [p. 22] and XML.

Member Function Documentation

QDomDocumentType::QDomDocumentType ()

Creates an empty QDomDocumentType object.

QDomDocumentType::QDomDocumentType (const QDomDocumentType & n)

Constructs a copy of *n*.

The data of the copy is shared (shallow copy): modifying one node will also change the other. If you want to make a deep copy, use cloneNode().

QDomDocumentType::~QDomDocumentType ()

Destroys the object and frees its resources.

QDomNamedNodeMap QDomDocumentType::entities () const [virtual]

Returns a map of all entities described in the DTD.

QString QDomDocumentType::internalSubset () const [virtual]

Returns the internal subset of the document type, if there is any. Otherwise this function returns QString::null.

See also publicId() [p. 35] and systemId() [p. 35].

bool QDomDocumentType::isDocumentType () const [virtual]

This function overloads QDomNode::isDocumentType().

See also nodeType() [p. 34] and QDomNode::toDocumentType() [p. 66].

Reimplemented from QDomNode [p. 59].

QString QDomDocumentType::name () const [virtual]

Returns the name of the document type as specified in the <!DOCTYPE name> tag.

See also nodeName() [p. 62].

QDomNode::NodeType QDomDocumentType::nodeType () const [virtual]

Returns DocumentTypeNode.

See also isDocumentType() [p. 34] and QDomNode::toDocumentType() [p. 66].

Reimplemented from QDomNode [p. 62].

QDomNamedNodeMap QDomDocumentType::notations () const [virtual]

Returns a map of all notations described in the DTD.

QDomDocumentType & QDomDocumentType::operator= (const QDomDocumentType & n)

Assigns *n* to this document type.

The data of the copy is shared (shallow copy): modifying one node will also change the other. If you want to make a deep copy, use cloneNode().

QString QDomDocumentType::publicId () const [virtual]

Returns the public identifier of the external DTD subset, if there is any. Otherwise this function returns QString::null.

See also systemId() [p. 35], internalSubset() [p. 34] and QDomImplementation::createDocumentType() [p. 47].

QString QDomDocumentType::systemId () const [virtual]

Returns the system identifier of the external DTD subset, if there is any. Otherwise this function returns QString::null.

See also publicId() [p. 35], internalSubset() [p. 34] and QDomImplementation::createDocumentType() [p. 47].

QDomElement Class Reference

The QDomElement class represents one element in the DOM tree.

This class is part of the **XML** module.

`#include <qdom.h>`

Inherits QDomNode [p. 53].

Public Members

- **QDomElement** ()
- **QDomElement** (const QDomElement & x)
- QDomElement & **operator=** (const QDomElement & x)
- **~QDomElement** ()
- QString **attribute** (const QString & name, const QString & defValue = QString::null) const
- void **setAttribute** (const QString & name, const QString & value)
- void **setAttribute** (const QString & name, int value)
- void **setAttribute** (const QString & name, uint value)
- void **setAttribute** (const QString & name, double value)
- void **removeAttribute** (const QString & name)
- QDomAttr **attributeNode** (const QString & name)
- QDomAttr **setAttributeNode** (const QDomAttr & newAttr)
- QDomAttr **removeAttributeNode** (const QDomAttr & oldAttr)
- virtual QDomNodeList **elementsByTagName** (const QString & tagname) const
- bool **hasAttribute** (const QString & name) const
- QString **attributeNS** (const QString nsURI, const QString & localName, const QString & defValue) const
- void **setAttributeNS** (const QString nsURI, const QString & qName, const QString & value)
- void **setAttributeNS** (const QString nsURI, const QString & qName, int value)
- void **setAttributeNS** (const QString nsURI, const QString & qName, uint value)
- void **setAttributeNS** (const QString nsURI, const QString & qName, double value)
- void **removeAttributeNS** (const QString & nsURI, const QString & localName)
- QDomAttr **attributeNodeNS** (const QString & nsURI, const QString & localName)
- QDomAttr **setAttributeNodeNS** (const QDomAttr & newAttr)
- virtual QDomNodeList **elementsByTagNameNS** (const QString & nsURI, const QString & localName) const
- bool **hasAttributeNS** (const QString & nsURI, const QString & localName) const
- QString **tagName** () const
- void **setTagName** (const QString & name)
- virtual QDomNamedNodeMap **attributes** () const
- virtual QDomNode::NodeType **nodeType** () const
- virtual bool **isElement** () const
- QString **text** () const

Detailed Description

The QDomElement class represents one element in the DOM tree.

Elements have a tagName() and zero or more attributes associated with them. The tag name can be changed with setTagName().

Attributes of the element are represented by QDomAttr objects, that can be queried using the attribute() and attributeNode() functions. You can set attributes with the setAttribute() and setAttributeNode() functions. Attributes can be removed with removeAttribute(). There are namespace-aware equivalents to these functions, i.e. setAttributeNS(), setAttributeNodeNS() and removeAttributeNS().

If you want to access the text of a node use text(), e.g.

```
    QDomElement e = //...
    //...
    QString s = e.text()
```

The text() function operates recursively to find the text (since not all elements contain text). If you want to find all the text in all of a node's children iterate over the children looking for QDomText nodes, e.g.

```
    QString text;
    QDomElement element = doc.documentElement();
    for( QDomNode n = element.firstChild(); !n.isNull(); n = n.nextSibling() )
    {
            QDomText t = n.toText();
            if ( !t.isNull() )
                    text += t.data();
    }
```

Note that we attempt to convert each node to a text node and use text() rather than using firstChild().toText().data() or n.toText().data() directly on the node, because the node may not be a text element.

You can get a list of all the decendents of an element which have a specified tag name with elementsByTagName() or elementsByTagNameNS().

For further information about the Document Object Model see http://www.w3.org/TR/REC-DOM-Level-1/ and http://www.w3.org/TR/DOM-Level-2-Core/. For a more general introduction of the DOM implementation see the QDomDocument documentation.

See also XML.

Member Function Documentation

QDomElement::QDomElement ()

Constructs an empty element. Use the QDomDocument::createElement() function to construct elements with content.

QDomElement::QDomElement (const QDomElement & x)

Constructs a copy of *x*.

The data of the copy is shared (shallow copy): modifying one node will also change the other. If you want to make a deep copy, use cloneNode().

QDomElement::~QDomElement ()

Destroys the object and frees its resources.

QString QDomElement::attribute (const QString & name, const QString & defValue = QString::null) const

Returns the attribute called *name*. If the attribute does not exist *defValue* is returned.

See also setAttribute() [p. 40], attributeNode() [p. 38], setAttributeNode() [p. 41] and attributeNS() [p. 38].

QString QDomElement::attributeNS (const QString nsURI, const QString & localName, const QString & defValue) const

Returns the attribute with the local name *localName* and the namespace URI *nsURI*. If the attribute does not exist *defValue* is returned.

See also setAttributeNS() [p. 40], attributeNodeNS() [p. 38], setAttributeNodeNS() [p. 41] and attribute() [p. 38].

QDomAttr QDomElement::attributeNode (const QString & name)

Returns the QDomAttr object that corresponds to the attribute called *name*. If no such attribute exists a null object is returned.

See also setAttributeNode() [p. 41], attribute() [p. 38], setAttribute() [p. 40] and attributeNodeNS() [p. 38].

QDomAttr QDomElement::attributeNodeNS (const QString & nsURI, const QString & localName)

Returns the QDomAttr object that corresponds to the attribute with the local name *localName* and the namespace URI *nsURI*. If no such attribute exists a null object is returned.

See also setAttributeNode() [p. 41], attribute() [p. 38] and setAttribute() [p. 40].

QDomNamedNodeMap QDomElement::attributes () const [virtual]

Returns a QDomNamedNodeMap containing all this element's attributes.

See also attribute() [p. 38], setAttribute() [p. 40], attributeNode() [p. 38] and setAttributeNode() [p. 41].

Reimplemented from QDomNode [p. 57].

QDomNodeList QDomElement::elementsByTagName (const QString & tagname) const [virtual]

Returns a QDomNodeList containing all descendant elements of this element called *tagname*. The order they are in the node list, is the order they are encountered in a preorder traversal of the element tree.

See also elementsByTagNameNS() [p. 39] and QDomDocument::elementsByTagName() [p. 27].

QDomNodeList QDomElement::elementsByTagNameNS (const QString & nsURI, const QString & localName) const [virtual]

Returns a QDomNodeList containing all the descendant elements of this element with the local name *localName* and the namespace URI *nsURI*. The order they are in the node list, is the order they are encountered in a preorder traversal of the element tree.

See also elementsByTagName() [p. 38] and QDomDocument::elementsByTagNameNS() [p. 27].

bool QDomElement::hasAttribute (const QString & name) const

Returns TRUE if this element has an attribute called *name*; otherwise returns FALSE.

bool QDomElement::hasAttributeNS (const QString & nsURI, const QString & localName) const

Returns TRUE if this element has an attribute with the local name *localName* and the namespace URI *nsURI*; otherwise returns FALSE.

bool QDomElement::isElement () const [virtual]

Returns TRUE.

Reimplemented from QDomNode [p. 59].

QDomNode::NodeType QDomElement::nodeType () const [virtual]

Returns ElementNode.

Reimplemented from QDomNode [p. 62].

QDomElement & QDomElement::operator= (const QDomElement & x)

Assigns *x* to this DOM element.

The data of the copy is shared (shallow copy): modifying one node will also change the other. If you want to make a deep copy, use cloneNode().

void QDomElement::removeAttribute (const QString & name)

Removes the attribute called name *name* from this element.

See also setAttribute() [p. 40], attribute() [p. 38] and removeAttributeNS() [p. 39].

void QDomElement::removeAttributeNS (const QString & nsURI, const QString & localName)

Removes the attribute with the local name *localName* and the namespace URI *nsURI* from this element.

See also setAttributeNS() [p. 40], attributeNS() [p. 38] and removeAttribute() [p. 39].

QDomAttr QDomElement::removeAttributeNode (const QDomAttr & oldAttr)

Removes the attribute *oldAttr* from the element and returns it.

See also attributeNode() [p. 38] and setAttributeNode() [p. 41].

void QDomElement::setAttribute (const QString & name, const QString & value)

Adds an attribute called name *name* with value *value*. If an attribute with the same name exists, its value is replaced by *value*.

See also attribute() [p. 38], setAttributeNode() [p. 41] and setAttributeNS() [p. 40].

void QDomElement::setAttribute (const QString & name, int value)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

void QDomElement::setAttribute (const QString & name, uint value)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

void QDomElement::setAttribute (const QString & name, double value)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

void QDomElement::setAttributeNS (const QString nsURI, const QString & qName, const QString & value)

Adds an attribute with the qualified name *qName* and the namespace URI *nsURI* with the value *value*. If an attribute with the same local name and namespace URI exists, its prefix is replaced by the prefix of *qName* and its value is repaced by *value*.

Although *qName* is the qualified name, the local name is used to decide if an existing attribute's value should be replaced.

See also attributeNS() [p. 38], setAttributeNodeNS() [p. 41] and setAttribute() [p. 40].

void QDomElement::setAttributeNS (const QString nsURI, const QString & qName, int value)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

void QDomElement::setAttributeNS (const QString nsURI, const QString & qName, uint value)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

void QDomElement::setAttributeNS (const QString nsURI, const QString & qName, double value)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

QDomAttr QDomElement::setAttributeNode (const QDomAttr & newAttr)

Adds the attribute *newAttr* to this element.

If the element has another attribute that has the same name as *newAttr*, this function replaces that attribute and returns it; otherwise the function returns a null attribute.

See also attributeNode() [p. 38], setAttribute() [p. 40] and setAttributeNodeNS() [p. 41].

QDomAttr QDomElement::setAttributeNodeNS (const QDomAttr & newAttr)

Adds the attribute *newAttr* to this element.

If the element has another attribute that has the same local name and namespace URI as *newAttr*, this function replaces that attribute and returns it; otherwise the function returns a null attribute.

See also attributeNodeNS() [p. 38], setAttributeNS() [p. 40] and setAttributeNode() [p. 41].

void QDomElement::setTagName (const QString & name)

Sets the tag name of this element to *name*.

See also tagName() [p. 41].

QString QDomElement::tagName () const

Returns the tag name of this element. For an XML element like

the tagname would return "img".

See also setTagName() [p. 41].

QString QDomElement::text () const

Returns the text contained inside this element.

Example:

```
Hello Qt <![CDATA[]]>
```

The function text() of the QDomElement for the <h1> tag, will return "Hello Qt <xml is cool>".

Comments are ignored by this function. It only evaluates QDomText and QDomCDATASection objects.

QDomEntity Class Reference

The QDomEntity class represents an XML entity.

This class is part of the **XML** module.

`#include <qdom.h>`

Inherits QDomNode [p. 53].

Public Members

- **QDomEntity** ()
- **QDomEntity** (const QDomEntity & x)
- QDomEntity & **operator=** (const QDomEntity & x)
- **~QDomEntity** ()
- virtual QString **publicId** () const
- virtual QString **systemId** () const
- virtual QString **notationName** () const
- virtual QDomNode::NodeType **nodeType** () const
- virtual bool **isEntity** () const

Detailed Description

The QDomEntity class represents an XML entity.

This class represents an entity in an XML document, either parsed or unparsed. Note that this models the entity itself not the entity declaration.

DOM does not support editing entity nodes; if a user wants to make changes to the contents of an entity, every related QDomEntityReference node has to be replaced in the DOM tree by a clone of the entity's contents, and then the desired changes must be made to each of the clones instead. All the descendants of an entity node are read-only.

An entity node does not have any parent.

You can access the entity's publicId(), systemId() and notationName() when available.

For further information about the Document Object Model see http://www.w3.org/TR/REC-DOM-Level-1/ and http://www.w3.org/TR/DOM-Level-2-Core/. For a more general introduction of the DOM implementation see the QDomDocument documentation.

See also XML.

Member Function Documentation

QDomEntity::QDomEntity ()

Constructs an empty entity.

QDomEntity::QDomEntity (const QDomEntity & x)

Constructs a copy of *x*.

The data of the copy is shared (shallow copy): modifying one node will also change the other. If you want to make a deep copy, use cloneNode().

QDomEntity::~QDomEntity ()

Destroys the object and frees its resources.

bool QDomEntity::isEntity () const [virtual]

Returns TRUE.

Reimplemented from QDomNode [p. 60].

QDomNode::NodeType QDomEntity::nodeType () const [virtual]

Returns EntityNode.

Reimplemented from QDomNode [p. 62].

QString QDomEntity::notationName () const [virtual]

For unparsed entities this function returns the name of the notation for the entity. For parsed entities this function returns QString::null.

QDomEntity & QDomEntity::operator= (const QDomEntity & x)

Assigns *x* to this DOM entity.

The data of the copy is shared (shallow copy): modifying one node will also change the other. If you want to make a deep copy, use cloneNode().

QString QDomEntity::publicId () const [virtual]

Returns the public identifier associated with this entity. If the public identifier was not specified QString::null is returned.

QString QDomEntity::systemId () const [virtual]

Returns the system identifier associated with this entity. If the system identifier was not specified QString::null is returned.

QDomEntityReference Class Reference

The QDomEntityReference class represents an XML entity reference.

This class is part of the **XML** module.

`#include <qdom.h>`

Inherits QDomNode [p. 53].

Public Members

- **QDomEntityReference** ()
- **QDomEntityReference** (const QDomEntityReference & x)
- QDomEntityReference & **operator=** (const QDomEntityReference & x)
- **~QDomEntityReference** ()
- virtual QDomNode::NodeType **nodeType** () const
- virtual bool **isEntityReference** () const

Detailed Description

The QDomEntityReference class represents an XML entity reference.

A QDomEntityReference object may be inserted into the DOM tree when an entity reference is in the source document, or when the user wishes to insert an entity reference.

Note that character references and references to predefined entities are expanded by the XML processor so that characters are represented by their Unicode equivalent rather than by an entity reference.

Moreover, the XML processor may completely expand references to entities while building the DOM tree, instead of providing QDomEntityReference objects.

If it does provide such objects, then for a given entity reference node, it may be that there is no entity node representing the referenced entity; but if such an entity exists, then the child list of the entity reference node is the same as that of the entity node. As with the entity node, all descendants of the entity reference are read-only.

For further information about the Document Object Model see http://www.w3.org/TR/REC-DOM-Level-1/ and http://www.w3.org/TR/DOM-Level-2-Core/. For a more general introduction of the DOM implementation see the QDomDocument documentation.

See also XML.

Member Function Documentation

QDomEntityReference::QDomEntityReference ()

Constructs an empty entity reference. Use QDomDocument::createEntityReference() to create a entity reference with content.

QDomEntityReference::QDomEntityReference (const QDomEntityReference & x)

Constructs a copy of *x*.

The data of the copy is shared (shallow copy): modifying one node will also change the other. If you want to make a deep copy, use cloneNode().

QDomEntityReference::~QDomEntityReference ()

Destroys the object and frees its resources.

bool QDomEntityReference::isEntityReference () const [virtual]

Returns TRUE.

Reimplemented from QDomNode [p. 60].

QDomNode::NodeType QDomEntityReference::nodeType () const [virtual]

Returns `EntityReference`.

Reimplemented from QDomNode [p. 62].

QDomEntityReference & QDomEntityReference::operator= (const QDomEntityReference & x)

Assigns *x* to this entity reference.

The data of the copy is shared (shallow copy): modifying one node will also change the other. If you want to make a deep copy, use cloneNode().

QDomImplementation Class Reference

The QDomImplementation class provides information about the features of the DOM implementation.

This class is part of the **XML** module.

`#include <qdom.h>`

Public Members

- **QDomImplementation** ()
- **QDomImplementation** (const QDomImplementation & x)
- virtual **~QDomImplementation** ()
- QDomImplementation & **operator=** (const QDomImplementation & x)
- bool **operator==** (const QDomImplementation & x) const
- bool **operator!=** (const QDomImplementation & x) const
- virtual bool **hasFeature** (const QString & feature, const QString & version)
- virtual QDomDocumentType **createDocumentType** (const QString & qName, const QString & publicId, const QString & systemId)
- virtual QDomDocument **createDocument** (const QString & nsURI, const QString & qName, const QDomDocumentType & doctype)
- bool **isNull** ()

Detailed Description

The QDomImplementation class provides information about the features of the DOM implementation.

This class describes the features that are supported by the DOM implementation. Currently only the XML subset of DOM Level 1 and DOM Level 2 Core are supported.

Normally you will use the function QDomDocument::implementation() to get the implementation object.

You can create a new document type with createDocumentType() and a new document with createDocument().

For further information about the Document Object Model see http://www.w3.org/TR/REC-DOM-Level-1/ and http://www.w3.org/TR/DOM-Level-2-Core/. For a more general introduction of the DOM implementation see the QDomDocument documentation.

See also hasFeature() [p. 47] and XML.

Member Function Documentation

QDomImplementation::QDomImplementation ()

Constructs a QDomImplementation object.

QDomImplementation::QDomImplementation (const QDomImplementation & x)

Constructs a copy of *x*.

QDomImplementation::~QDomImplementation () [virtual]

Destroys the object and frees its resources.

QDomDocument QDomImplementation::createDocument (const QString & nsURI, const QString & qName, const QDomDocumentType & doctype) [virtual]

Creates a DOM document with the document type *doctype*. This function also adds a root element node with the qualified name *qName* and the namespace URI *nsURI*.

QDomDocumentType QDomImplementation::createDocumentType (const QString & qName, const QString & publicId, const QString & systemId) [virtual]

Creates a document type node for the name *qName*.

publicId specifies the public identifier of the external subset; If you specify QString::null as the *publicId*, this means that the document type has no public identifier.

Similarly, you specify the system identifier of the external subset with *systemId*. If you specify QString::null as the *systemId*, this means that the document type has no system identifier. Since you cannot have a public identifier without a system identifier, the public identifier is set to QString::null if there is no system identifier.

Other features of a document type declaration are not supported by DOM level 2.

The only way you can use a document type that was created this way, is in combination with the createDocument() function to create a QDomDocument with this document type.

See also createDocument() [p. 47].

bool QDomImplementation::hasFeature (const QString & feature, const QString & version) [virtual]

The function returns TRUE if QDom implements the requested *version* of a *feature*.

Currently only the feature "XML" in version "1.0" is supported.

bool QDomImplementation::isNull ()

Returns FALSE if the object was created by QDomDocument::implementation(); otherwise returns TRUE.

bool QDomImplementation::operator!= (const QDomImplementation & x) const

Returns TRUE if *x* and this DOM implementation object were created from different QDomDocuments.

QDomImplementation & QDomImplementation::operator= (const QDomImplementation & x)

Assigns *x* to this DOM implementation.

bool QDomImplementation::operator== (const QDomImplementation & x) const

Returns TRUE if *x* and this DOM implementation object were created from the same QDomDocument.

QDomNamedNodeMap Class Reference

The QDomNamedNodeMap class contains a collection of nodes that can be accessed by name.

This class is part of the **XML** module.

```
#include <qdom.h>
```

Public Members

- **QDomNamedNodeMap** ()
- **QDomNamedNodeMap** (const QDomNamedNodeMap & n)
- QDomNamedNodeMap & **operator=** (const QDomNamedNodeMap & n)
- bool **operator==** (const QDomNamedNodeMap & n) const
- bool **operator!=** (const QDomNamedNodeMap & n) const
- **~QDomNamedNodeMap** ()
- QDomNode **namedItem** (const QString & name) const
- QDomNode **setNamedItem** (const QDomNode & newNode)
- QDomNode **removeNamedItem** (const QString & name)
- QDomNode **item** (int index) const
- QDomNode **namedItemNS** (const QString & nsURI, const QString & localName) const
- QDomNode **setNamedItemNS** (const QDomNode & newNode)
- QDomNode **removeNamedItemNS** (const QString & nsURI, const QString & localName)
- uint **length** () const
- uint **count** () const
- bool **contains** (const QString & name) const

Detailed Description

The QDomNamedNodeMap class contains a collection of nodes that can be accessed by name.

Note that QDomNamedNodeMap does not inherit from QDomNodeList. QDomNamedNodeMaps do not provide any specific node ordering. Although nodes in a QDomNamedNodeMap may be accessed by an ordinal index, this is simply to allow a convenient enumeration of the contents of a QDomNamedNodeMap, and does not imply that the DOM specifies an ordering of the nodes.

The QDomNamedNodeMap is used in three places:

- QDomDocumentType::entities() returns a map of all entities described in the DTD.
- QDomDocumentType::notations() returns a map of all notations described in the DTD.
- QDomNode::attributes() returns a map of all attributes of the element.

Items in the map are identified by the name which QDomNode::name() returns. Nodes are retrieved using namedItem(), namedItemNS() or item(). New nodes are inserted with setNamedItem() or setNamedItem() and removed with removeNamedItem() or removeNamedItemNS(). Use contains() to see if an item with the given name is in the named node map. The number of items is returned by length().

Terminology: in this class we use "item" and "node" interchangeably.

See also XML.

Member Function Documentation

QDomNamedNodeMap::QDomNamedNodeMap ()

Constructs an empty named node map.

QDomNamedNodeMap::QDomNamedNodeMap (const QDomNamedNodeMap & n)

Constructs a copy of *n*.

QDomNamedNodeMap::~QDomNamedNodeMap ()

Destroys the object and frees its resources.

bool QDomNamedNodeMap::contains (const QString & name) const

Returns TRUE if the map contains a node called *name*; otherwise returns FALSE.

uint QDomNamedNodeMap::count () const

Returns the number of nodes in the map.

This function is the same as length().

QDomNode QDomNamedNodeMap::item (int index) const

Retrieves the node at position *index*.

This can be used to iterate over the map. Note that the nodes in the map are ordered arbitrarily.

See also length() [p. 50].

uint QDomNamedNodeMap::length () const

Returns the number of nodes in the map.

See also item() [p. 50].

QDomNode QDomNamedNodeMap::namedItem (const QString & name) const

Returns the node called *name*.

If the named node map does not contain such a node, a null node is returned. A node's name is the name returned by QDomNode::nodeName().

See also setNamedItem() [p. 51] and namedItemNS() [p. 51].

QDomNode QDomNamedNodeMap::namedItemNS (const QString & nsURI, const QString & localName) const

Returns the node associated with the local name *localName* and the namespace URI *nsURI*.

If the map does not contain such a node, a null node is returned.

See also setNamedItemNS() [p. 52] and namedItem() [p. 50].

bool QDomNamedNodeMap::operator!= (const QDomNamedNodeMap & n) const

Returns TRUE if *n* and this named node map are not equal; otherwise returns FALSE.

QDomNamedNodeMap & QDomNamedNodeMap::operator= (const QDomNamedNodeMap & n)

Assigns *n* to this named node map.

bool QDomNamedNodeMap::operator== (const QDomNamedNodeMap & n) const

Returns TRUE if *n* and this named node map are equal; otherwise returns FALSE.

QDomNode QDomNamedNodeMap::removeNamedItem (const QString & name)

Removes the node called *name* from the map.

The function returns the removed node or a null node if the map did not contain a node called *name*.

See also setNamedItem() [p. 51], namedItem() [p. 50] and removeNamedItemNS() [p. 51].

QDomNode QDomNamedNodeMap::removeNamedItemNS (const QString & nsURI, const QString & localName)

Removes the node with the local name *localName* and the namespace URI *nsURI* from the map.

The function returns the removed node or a null node if the map did not contain a node with the local name *localName* and the namespace URI *nsURI*.

See also setNamedItemNS() [p. 52], namedItemNS() [p. 51] and removeNamedItem() [p. 51].

QDomNode QDomNamedNodeMap::setNamedItem (const QDomNode & newNode)

Inserts the node *newNode* into the named node map. The name used by the map is the node name of *newNode* as returned by QDomNode::nodeName().

If the new node replaces an existing node, i.e. the map contains a node with the same name, the replaced node is returned.

See also namedItem() [p. 50], removeNamedItem() [p. 51] and setNamedItemNS() [p. 52].

QDomNode QDomNamedNodeMap::setNamedItemNS (const QDomNode & newNode)

Inserts the node *newNode* in the map. If a node with the same namespace URI and the same local name already exists in the map, it is replaced by *newNode*. If the new node replaces an existing node, the replaced node is returned.

See also namedItemNS() [p. 51], removeNamedItemNS() [p. 51] and setNamedItem() [p. 51].

QDomNode Class Reference

The QDomNode class is the base class for all the nodes in a DOM tree.

This class is part of the **XML** module.

`#include <qdom.h>`

Inherited by QDomDocumentType [p. 33], QDomDocument [p. 22], QDomDocumentFragment [p. 31], QDomCharacterData [p. 17], QDomAttr [p. 12], QDomElement [p. 36], QDomNotation [p. 70], QDomEntity [p. 42], QDomEntityReference [p. 44] and QDomProcessingInstruction [p. 72].

Public Members

- enum **NodeType** { ElementNode = 1, AttributeNode = 2, TextNode = 3, CDATASectionNode = 4, EntityReferenceNode = 5, EntityNode = 6, ProcessingInstructionNode = 7, CommentNode = 8, DocumentNode = 9, DocumentTypeNode = 10, DocumentFragmentNode = 11, NotationNode = 12, BaseNode = 21, CharacterDataNode = 22 }
- **QDomNode** ()
- **QDomNode** (const QDomNode & n)
- QDomNode & **operator=** (const QDomNode & n)
- bool **operator==** (const QDomNode & n) const
- bool **operator!=** (const QDomNode & n) const
- virtual **~QDomNode** ()
- virtual QDomNode **insertBefore** (const QDomNode & newChild, const QDomNode & refChild)
- virtual QDomNode **insertAfter** (const QDomNode & newChild, const QDomNode & refChild)
- virtual QDomNode **replaceChild** (const QDomNode & newChild, const QDomNode & oldChild)
- virtual QDomNode **removeChild** (const QDomNode & oldChild)
- virtual QDomNode **appendChild** (const QDomNode & newChild)
- virtual bool **hasChildNodes** () const
- virtual QDomNode **cloneNode** (bool deep = TRUE) const
- virtual void **normalize** ()
- virtual bool **isSupported** (const QString & feature, const QString & version) const
- virtual QString **nodeName** () const
- virtual QDomNode::NodeType **nodeType** () const
- virtual QDomNode **parentNode** () const
- virtual QDomNodeList **childNodes** () const
- virtual QDomNode **firstChild** () const
- virtual QDomNode **lastChild** () const
- virtual QDomNode **previousSibling** () const
- virtual QDomNode **nextSibling** () const
- virtual QDomNamedNodeMap **attributes** () const
- virtual QDomDocument **ownerDocument** () const

- virtual QString **namespaceURI** () const
- virtual QString **localName** () const
- virtual bool **hasAttributes** () const
- virtual QString **nodeValue** () const
- virtual void **setNodeValue** (const QString & v)
- virtual QString **prefix** () const
- virtual void **setPrefix** (const QString & pre)
- virtual bool **isAttr** () const
- virtual bool **isCDATASection** () const
- virtual bool **isDocumentFragment** () const
- virtual bool **isDocument** () const
- virtual bool **isDocumentType** () const
- virtual bool **isElement** () const
- virtual bool **isEntityReference** () const
- virtual bool **isText** () const
- virtual bool **isEntity** () const
- virtual bool **isNotation** () const
- virtual bool **isProcessingInstruction** () const
- virtual bool **isCharacterData** () const
- virtual bool **isComment** () const
- QDomNode **namedItem** (const QString & name) const
- bool **isNull** () const
- void **clear** ()
- QDomAttr **toAttr** ()
- QDomCDATASection **toCDATASection** ()
- QDomDocumentFragment **toDocumentFragment** ()
- QDomDocument **toDocument** ()
- QDomDocumentType **toDocumentType** ()
- QDomElement **toElement** ()
- QDomEntityReference **toEntityReference** ()
- QDomText **toText** ()
- QDomEntity **toEntity** ()
- QDomNotation **toNotation** ()
- QDomProcessingInstruction **toProcessingInstruction** ()
- QDomCharacterData **toCharacterData** ()
- QDomComment **toComment** ()
- void **save** (QTextStream & str, int indent) const

Related Functions

- QTextStream & **operator<<** (QTextStream & str, const QDomNode & node)

Detailed Description

The QDomNode class is the base class for all the nodes in a DOM tree.

Many functions in the DOM return a QDomNode.

You can find out the type of a node using isAttr(), isCDATASection(), isDocumentFragment(), isDocument(), isDocumentType(), isElement(), isEntityReference(), isText(), isEntity(), isNotation(), isProcessingInstruction(), isCharacterData() and isComment().

A QDomNode can be converted into one of its subclasses using toAttr(), toCDATASection(), toDocumentFragment(), toDocument(), toDocumentType(), toElement(), toEntityReference(), toText(), toEntity(), toNotation(), toProcessingInstruction(), toCharacterData() or toComment(). You can convert a node to a null node with clear().

Copies of the QDomNode class share their data; this means modifying one node will change all copies. This is especially useful in combination with functions which return a QDomNode, e.g. firstChild(). You can make an independent (deep) copy of the node with cloneNode().

Nodes are inserted with insertBefore(), insertAfter() or appendChild(). You can replace one node with another using replaceChild() and remove a node with removeChild().

To traverse nodes use firstChild() to get a node's first child (if any), and nextSibling() to traverse. QDomNode also provides lastChild(), previousSibling() and parentNode(). To find the first child node with a particular node name use namedItem().

To find out if a node has children use hasChildNodes() and to get a list of all of a node's children use childNodes().

The node's name and value (the meaning of which varies depending on its type) is returned by nodeName() and nodeValue() respectively. The node's type is returned by nodeType(). The node's value can be set with setNodeValue().

The document to which the node belongs is returned by ownerDocument().

Adjacent QDomText nodes can be merged into a single node with normalize().

QDomElement nodes have attributes which can be retrieved with attributes().

QDomElement and QDomAttr nodes can have namespaces which can be retrieved with namespaceURI(). Their local name is retrieved with localName(), and their prefix with prefix(). The prefix can be set with setPrefix().

You can write the XML representation of the node to a text stream with save().

The following example looks for the first element in an XML document and prints the names of all the elements that are its direct children.

```
QDomDocument d;
d.setContent( someXML );
QDomNode n = d.firstChild();
while ( !n.isNull() ) {
    if ( n.isElement() ) {
        QDomElement e = n.toElement();
        cout << "Element name: " << e.tagName() << endl;
        return;
    }
    n = n.nextSibling();
}
```

For further information about the Document Object Model see http://www.w3.org/TR/REC-DOM-Level-1/ and http://www.w3.org/TR/DOM-Level-2-Core/. For a more general introduction of the DOM implementation see the QDomDocument documentation.

See also XML.

Member Type Documentation

QDomNode::NodeType

This enum defines the type of the node:

- `QDomNode::ElementNode`
- `QDomNode::AttributeNode`
- `QDomNode::TextNode`
- `QDomNode::CDATASectionNode`
- `QDomNode::EntityReferenceNode`
- `QDomNode::EntityNode`
- `QDomNode::ProcessingInstructionNode`
- `QDomNode::CommentNode`
- `QDomNode::DocumentNode`
- `QDomNode::DocumentTypeNode`
- `QDomNode::DocumentFragmentNode`
- `QDomNode::NotationNode`
- `QDomNode::BaseNode` - A QDomNode object, i.e. not a QDomNode subclass.
- `QDomNode::CharacterDataNode`

Member Function Documentation

QDomNode::QDomNode ()

Constructs an empty node.

QDomNode::QDomNode (const QDomNode & n)

Constructs a copy of *n*.

The data of the copy is shared (shallow copy): modifying one node will also change the other. If you want to make a deep copy, use cloneNode().

QDomNode::~QDomNode () [virtual]

Destroys the object and frees its resources.

QDomNode QDomNode::appendChild (const QDomNode & newChild) [virtual]

Appends *newChild* as the node's last child.

If *newChild* is the child of another node, it is reparented to this node. If *newChild* is a child of this node, then its position in the list of children is changed.

Returns a new reference to *newChild*.

See also insertBefore() [p. 58], insertAfter() [p. 58], replaceChild() [p. 64] and removeChild() [p. 64].

QDomNamedNodeMap QDomNode::attributes () const [virtual]

Returns a named node map of all attributes. Attributes are only provided for QDomElement.

Changing the attributes in the map will also change the attributes of this QDomNode.

Reimplemented in QDomElement.

QDomNodeList QDomNode::childNodes () const [virtual]

Returns a list of all direct child nodes.

Most often you will call this function on a QDomElement object.

For example, if the XML document looks like this:

```
Heading
Hello you
```

Then the list of child nodes for the "body"-element will contain the node created by the <h1> tag and the node created by the <p> tag.

The nodes in the list are not copied; so changing the nodes in the list will also change the children of this node.

See also firstChild() [p. 57] and lastChild() [p. 61].

void QDomNode::clear ()

Dereferences the internal object. The node becomes a null node.

See also isNull() [p. 60].

QDomNode QDomNode::cloneNode (bool deep = TRUE) const [virtual]

Creates a real (not shallow) copy of the QDomNode.

If *deep* is TRUE, then the cloning is done recursively which means that all the node's children are copied, too. If *deep* is FALSE only the node itself is copied and the copy will have no child nodes.

QDomNode QDomNode::firstChild () const [virtual]

Returns the first child of the node. If there is no child node, a null node is returned. Changing the returned node will also change the node in the document tree.

See also lastChild() [p. 61] and childNodes() [p. 57].

Example: xml/outliner/outlinetree.cpp.

bool QDomNode::hasAttributes () const [virtual]

Returns TRUE if the node has attributes; otherwise returns FALSE.

See also attributes() [p. 57].

bool QDomNode::hasChildNodes () const [virtual]

Returns TRUE if the node has one or more children; otherwise returns FALSE.

QDomNode QDomNode::insertAfter (const QDomNode & newChild, const QDomNode & refChild) [virtual]

Inserts the node *newChild* after the child node *refChild*. *refChild* must be a direct child of this node. If *refChild* is null then *newChild* is appended as this node's last child.

If *newChild* is the child of another node, it is reparented to this node. If *newChild* is a child of this node, then its position in the list of children is changed.

If *newChild* is a QDomDocumentFragment, then the children of the fragment are removed from the fragment and inserted after *refChild*.

Returns a new reference to *newChild* on success or an empty node on failure.

See also insertBefore() [p. 58], replaceChild() [p. 64], removeChild() [p. 64] and appendChild() [p. 56].

QDomNode QDomNode::insertBefore (const QDomNode & newChild, const QDomNode & refChild) [virtual]

Inserts the node *newChild* before the child node *refChild*. *refChild* must be a direct child of this node. If *refChild* is null then *newChild* is inserted as the node's first child.

If *newChild* is the child of another node, it is reparented to this node. If *newChild* is a child of this node, then its position in the list of children is changed.

If *newChild* is a QDomDocumentFragment, then the children of the fragment are removed from the fragment and inserted before *refChild*.

Returns a new reference to *newChild* on success or an empty node on failure.

See also insertAfter() [p. 58], replaceChild() [p. 64], removeChild() [p. 64] and appendChild() [p. 56].

bool QDomNode::isAttr () const [virtual]

Returns TRUE if the node is an attribute; otherwise returns FALSE.

If this function returns TRUE, it does not imply that this object is a QDomAttribute; you can get the QDomAttribute with toAttribute().

See also toAttr() [p. 65].

Reimplemented in QDomAttr.

bool QDomNode::isCDATASection () const [virtual]

Returns TRUE if the node is a CDATA section; otherwise returns FALSE.

If this function returns TRUE, it does not imply that this object is a QDomCDATASection; you can get the QDomCDATASection with toCDATASection().

See also toCDATASection() [p. 65].

Reimplemented in QDomCDATASection.

bool QDomNode::isCharacterData () const [virtual]

Returns TRUE if the node is a character data node; otherwise returns FALSE.

If this function returns TRUE, it does not imply that this object is a QDomCharacterData; you can get the QDom-CharacterData with toCharacterData().

See also toCharacterData() [p. 65].

Reimplemented in QDomCharacterData.

bool QDomNode::isComment () const [virtual]

Returns TRUE if the node is a comment; otherwise returns FALSE.

If this function returns TRUE, it does not imply that this object is a QDomComment; you can get the QDomComment with toComment().

See also toComment() [p. 65].

Reimplemented in QDomComment.

bool QDomNode::isDocument () const [virtual]

Returns TRUE if the node is a document; otherwise returns FALSE.

If this function returns TRUE, it does not imply that this object is a QDomDocument; you can get the QDomDocument with toDocument().

See also toDocument() [p. 65].

Reimplemented in QDomDocument.

bool QDomNode::isDocumentFragment () const [virtual]

Returns TRUE if the node is a document fragment; otherwise returns FALSE.

If this function returns TRUE, it does not imply that this object is a QDomDocumentFragment; you can get the QDomDocumentFragment with toDocumentFragment().

See also toDocumentFragment() [p. 65].

Reimplemented in QDomDocumentFragment.

bool QDomNode::isDocumentType () const [virtual]

Returns TRUE if the node is a document type; otherwise returns FALSE.

If this function returns TRUE, it does not imply that this object is a QDomDocumentType; you can get the QDom-DocumentType with toDocumentType().

See also toDocumentType() [p. 66].

Reimplemented in QDomDocumentType.

bool QDomNode::isElement () const [virtual]

Returns TRUE if the node is an element; otherwise returns FALSE.

If this function returns TRUE, it does not imply that this object is a QDomElement; you can get the QDomElement with toElement().

See also toElement() [p. 66].

Example: xml/outliner/outlinetree.cpp.

Reimplemented in QDomElement.

bool QDomNode::isEntity () const [virtual]

Returns TRUE if the node is an entity; otherwise returns FALSE.

If this function returns TRUE, it does not imply that this object is a QDomEntity; you can get the QDomEntity with toEntity().

See also toEntity() [p. 66].

Reimplemented in QDomEntity.

bool QDomNode::isEntityReference () const [virtual]

Returns TRUE if the node is an entity reference; otherwise returns FALSE.

If this function returns TRUE, it does not imply that this object is a QDomEntityReference; you can get the QDomEntityReference with toEntityReference().

See also toEntityReference() [p. 66].

Reimplemented in QDomEntityReference.

bool QDomNode::isNotation () const [virtual]

Returns TRUE if the node is a notation; otherwise returns FALSE.

If this function returns TRUE, it does not imply that this object is a QDomNotation; you can get the QDomNotation with toNotation().

See also toNotation() [p. 66].

Reimplemented in QDomNotation.

bool QDomNode::isNull () const

Returns TRUE if this node does not reference any internal object; otherwise returns FALSE.

Example: xml/outliner/outlinetree.cpp.

bool QDomNode::isProcessingInstruction () const [virtual]

Returns TRUE if the node is a processing instruction; otherwise returns FALSE.

If this function returns TRUE, it does not imply that this object is a QDomProcessingInstruction; you can get the QProcessingInstruction with toProcessingInstruction().

See also toProcessingInstruction() [p. 66].

Reimplemented in QDomProcessingInstruction.

bool QDomNode::isSupported (const QString & feature, const QString & version) const [virtual]

Returns TRUE if the DOM implementation implements the feature *feature* and that feature is supported by this node in the version *version*. Otherwise this function returns FALSE.

See also QDomImplementation::hasFeature() [p. 47].

bool QDomNode::isText () const [virtual]

Returns TRUE if the node is a text; otherwise returns FALSE.

If this function returns TRUE, it does not imply that this object is a QDomText; you can get the QDomText with toText().

See also toText() [p. 66].

Reimplemented in QDomText.

QDomNode QDomNode::lastChild () const [virtual]

Returns the last child of the node. If there is no child node, a null node is returned. Changing the returned node will also change the node in the document tree.

See also firstChild() [p. 57] and childNodes() [p. 57].

QString QDomNode::localName () const [virtual]

If the node uses namespaces, this function returns the local name of the node. Otherwise it returns QString::null.

Only nodes of type ElementNode or AttributeNode can have namespaces. A namespace must have been specified at creation time; it is not possible to add a namespace afterwards.

See also prefix() [p. 64], namespaceURI() [p. 61], QDomDocument::createElementNS() [p. 26] and QDomDocument::createAttributeNS() [p. 25].

QDomNode QDomNode::namedItem (const QString & name) const

Returns the first direct child node for which nodeName() equals *name*.

If no such direct child exists, a null node is returned.

See also nodeName() [p. 62].

QString QDomNode::namespaceURI () const [virtual]

Returns the namespace URI of this node; if the node has no namespace URI, this function returns QString::null.

Only nodes of type ElementNode or AttributeNode can have namespaces. A namespace URI must be specified at creation time and cannot be changed later.

See also prefix() [p. 64], localName() [p. 61], QDomDocument::createElementNS() [p. 26] and QDomDocument::createAttributeNS() [p. 25].

QDomNode QDomNode::nextSibling () const [virtual]

Returns the next sibling in the document tree. Changing the returned node will also change the node in the document tree.

If you have XML like this:

```
Heading
The text...
Next heading
```

and this QDomNode represents the <p> tag, nextSibling() will return the node representing the <h2> tag.

See also previousSibling() [p. 64].

Example: xml/outliner/outlinetree.cpp.

QString QDomNode::nodeName () const [virtual]

Returns the name of the node.

The meaning of the name depends on the subclass:

- QDomAttr - the name of the attribute
- QDomCDATASection - the string "#cdata-section"
- QDomComment - the string "#comment"
- QDomDocument - the string "#document"
- QDomDocumentFragment - the string "#document-fragment"
- QDomDocumentType - the name of the document type
- QDomElement - the tag name
- QDomEntity - the name of the entity
- QDomEntityReference - the name of the referenced entity
- QDomNotation - the name of the notation
- QDomProcessingInstruction - the target of the processing instruction
- QDomText - the string "#text"

See also nodeValue() [p. 63].

Example: xml/outliner/outlinetree.cpp.

QDomNode::NodeType QDomNode::nodeType () const [virtual]

Returns the type of the node.

See also toAttr() [p. 65], toCDATASection() [p. 65], toDocumentFragment() [p. 65], toDocument() [p. 65], toDocumentType() [p. 66], toElement() [p. 66], toEntityReference() [p. 66], toText() [p. 66], toEntity() [p. 66], toNotation() [p. 66], toProcessingInstruction() [p. 66], toCharacterData() [p. 65] and toComment() [p. 65].

Reimplemented in QDomDocumentType, QDomDocument, QDomDocumentFragment, QDomCharacterData, QDomAttr, QDomElement, QDomNotation, QDomEntity, QDomEntityReference and QDomProcessingInstruction.

QString QDomNode::nodeValue () const [virtual]

Returns the value of the node.

The meaning of the value depends on the subclass:

- QDomAttr - the attribute value
- QDomCDATASection - the content of the CDATA section
- QDomComment - the comment
- QDomProcessingInstruction - the data of the processing intruction
- QDomText - the text

All other subclasses do not have a node value and will return a null string.

See also setNodeValue() [p. 65] and nodeName() [p. 62].

Example: xml/outliner/outlinetree.cpp.

void QDomNode::normalize () [virtual]

Calling normalize() on an element converts all its children into a standard form. This means, that adjacent QDomText objects will be merged into a single text object (QDomCDATASection nodes are not merged).

bool QDomNode::operator!= (const QDomNode & n) const

Returns TRUE if *n* and this DOM node are not equal; otherwise returns FALSE.

QDomNode & QDomNode::operator= (const QDomNode & n)

Assigns a copy of *n* to this DOM node.

The data of the copy is shared (shallow copy): modifying one node will also change the other. If you want to make a deep copy, use cloneNode().

bool QDomNode::operator== (const QDomNode & n) const

Returns TRUE if *n* and this DOM node are equal; otherwise returns FALSE.

QDomDocument QDomNode::ownerDocument () const [virtual]

Returns the document to which this node belongs.

QDomNode QDomNode::parentNode () const [virtual]

Returns the parent node. If this node has no parent, a null node is returned (i.e. a node for which isNull() returns TRUE).

QString QDomNode::prefix () const [virtual]

Returns the namespace prefix of the node; if a node has no namespace prefix, this function returns QString::null.

Only nodes of type ElementNode or AttributeNode can have namespaces. A namespace prefix must be specified at creation time. If a node was created with a namespace prefix, you can change it later with setPrefix().

If you create an element or attribute with QDomDocument::createElement() or QDomDocument::createAttribute(), the prefix will be null. If you use QDomDocument::createElementNS() or QDomDocument::createAttributeNS() instead, the prefix will not be null - although it might be an empty string if the name does not have a prefix.

See also setPrefix() [p. 65], localName() [p. 61], namespaceURI() [p. 61], QDomDocument::createElementNS() [p. 26] and QDomDocument::createAttributeNS() [p. 25].

QDomNode QDomNode::previousSibling () const [virtual]

Returns the previous sibling in the document tree. Changing the returned node will also change the node in the document tree.

For example, if you have XML like this:

```
Heading
The text...
Next heading
```

and this QDomNode represents the <p> tag, previousSibling() will return the node representing the <h1> tag.

See also nextSibling() [p. 62].

QDomNode QDomNode::removeChild (const QDomNode & oldChild) [virtual]

Removes *oldChild* from the list of children. *oldChild* must be a direct child of this node.

Returns a new reference to *oldChild* on success or a null node on failure.

See also insertBefore() [p. 58], insertAfter() [p. 58], replaceChild() [p. 64] and appendChild() [p. 56].

QDomNode QDomNode::replaceChild (const QDomNode & newChild, const QDomNode & oldChild) [virtual]

Replaces *oldChild* with *newChild*. *oldChild* must be a direct child of this node.

If *newChild* is the child of another node, it is reparented to this node. If *newChild* is a child of this node, then its position in the list of children is changed.

If *newChild* is a QDomDocumentFragment, then *oldChild* is replaced by all of the children of the fragment.

Returns a new reference to *oldChild* on success or a null node an failure.

See also insertBefore() [p. 58], insertAfter() [p. 58], removeChild() [p. 64] and appendChild() [p. 56].

void QDomNode::save (QTextStream & str, int indent) const

Writes the XML representation of the node and all its children to the stream *str*. This function uses *indent* as the amount of space to indent the node.

void QDomNode::setNodeValue (const QString & v) [virtual]

Sets the value of the node to *v*.

See also nodeValue() [p. 63].

void QDomNode::setPrefix (const QString & pre) [virtual]

If the node has a namespace prefix, this function changes the namespace prefix of the node to *pre*. Otherwise this function has no effect.

Only nodes of type ElementNode or AttributeNode can have namespaces. A namespace prefix must have be specified at creation time; it is not possible to add a namespace prefix afterwards.

See also prefix() [p. 64], localName() [p. 61], namespaceURI() [p. 61], QDomDocument::createElementNS() [p. 26] and QDomDocument::createAttributeNS() [p. 25].

QDomAttr QDomNode::toAttr ()

Converts a QDomNode into a QDomAttr. If the node is not an attribute, the returned object will be null.

See also isAttr() [p. 58].

QDomCDATASection QDomNode::toCDATASection ()

Converts a QDomNode into a QDomCDATASection. If the node is not a CDATA section, the returned object will be null.

See also isCDATASection() [p. 58].

QDomCharacterData QDomNode::toCharacterData ()

Converts a QDomNode into a QDomCharacterData. If the node is not a character data node the returned object will be null.

See also isCharacterData() [p. 59].

QDomComment QDomNode::toComment ()

Converts a QDomNode into a QDomComment. If the node is not a comment the returned object will be null.

See also isComment() [p. 59].

QDomDocument QDomNode::toDocument ()

Converts a QDomNode into a QDomDocument. If the node is not a document the returned object will be null.

See also isDocument() [p. 59].

QDomDocumentFragment QDomNode::toDocumentFragment ()

Converts a QDomNode into a QDomDocumentFragment. If the node is not a document fragment the returned object will be null.

See also isDocumentFragment() [p. 59].

QDomDocumentType QDomNode::toDocumentType ()

Converts a QDomNode into a QDomDocumentType. If the node is not a document type the returned object will be null.

See also isDocumentType() [p. 59].

QDomElement QDomNode::toElement ()

Converts a QDomNode into a QDomElement. If the node is not an element the returned object will be null.

See also isElement() [p. 59].

Example: xml/outliner/outlinetree.cpp.

QDomEntity QDomNode::toEntity ()

Converts a QDomNode into a QDomEntity. If the node is not an entity the returned object will be null.

See also isEntity() [p. 60].

QDomEntityReference QDomNode::toEntityReference ()

Converts a QDomNode into a QDomEntityReference. If the node is not an entity reference, the returned object will be null.

See also isEntityReference() [p. 60].

QDomNotation QDomNode::toNotation ()

Converts a QDomNode into a QDomNotation. If the node is not a notation the returned object will be null.

See also isNotation() [p. 60].

QDomProcessingInstruction QDomNode::toProcessingInstruction ()

Converts a QDomNode into a QDomProcessingInstruction. If the node is not a processing instruction the returned object will be null.

See also isProcessingInstruction() [p. 60].

QDomText QDomNode::toText ()

Converts a QDomNode into a QDomText. If the node is not a text, the returned object will be null.

See also isText() [p. 61].

Related Functions

QTextStream & operator<< (QTextStream & str, const QDomNode & node)

Writes the XML representation of the node *node* and all its children to the stream *str*.

QDomNodeList Class Reference

The QDomNodeList class is a list of QDomNode objects.

This class is part of the **XML** module.

```
#include <qdom.h>
```

Public Members

- **QDomNodeList** ()
- **QDomNodeList** (const QDomNodeList & n)
- QDomNodeList & **operator=** (const QDomNodeList & n)
- bool **operator==** (const QDomNodeList & n) const
- bool **operator!=** (const QDomNodeList & n) const
- virtual **~QDomNodeList** ()
- virtual QDomNode **item** (int index) const
- virtual uint **length** () const
- uint **count** () const

Detailed Description

The QDomNodeList class is a list of QDomNode objects.

Lists can be obtained by QDomDocument::elementsByTagName() and QDomNode::childNodes(). The Document Object Model (DOM) requires these lists to be "live": whenever you change the underlying document, the contents of the list will get updated.

You can get a particular node from the list with item(). The number of items in the list is returned by count() (and by length()).

For further information about the Document Object Model see http://www.w3.org/TR/REC-DOM-Level-1/ and http://www.w3.org/TR/DOM-Level-2-Core/. For a more general introduction of the DOM implementation see the QDomDocument documentation.

See also QDomNode::childNodes() [p. 57], QDomDocument::elementsByTagName() [p. 27] and XML.

Member Function Documentation

QDomNodeList::QDomNodeList ()

Creates an empty node list.

QDomNodeList::QDomNodeList (const QDomNodeList & n)

Constructs a copy of *n*.

QDomNodeList::~QDomNodeList () [virtual]

Destroys the object and frees its resources.

uint QDomNodeList::count () const

Returns the number of nodes in the list.

This function is the same as length().

QDomNode QDomNodeList::item (int index) const [virtual]

Returns the node at position *index*.

If *index* is negative or if *index* >= length() then a null node is returned (i.e. a node for which QDomNode::isNull() returns TRUE).

See also count() [p. 69].

uint QDomNodeList::length () const [virtual]

Returns the number of nodes in the list.

This function is the same as count().

bool QDomNodeList::operator!= (const QDomNodeList & n) const

Returns TRUE the node list *n* and this node list are not equal; otherwise returns FALSE.

QDomNodeList & QDomNodeList::operator= (const QDomNodeList & n)

Assigns *n* to this node list.

bool QDomNodeList::operator== (const QDomNodeList & n) const

Returns TRUE if the node list *n* and this node list are equal; otherwise returns FALSE.

QDomNotation Class Reference

The QDomNotation class represents an XML notation.

This class is part of the **XML** module.

`#include <qdom.h>`

Inherits QDomNode [p. 53].

Public Members

- **QDomNotation** ()
- **QDomNotation** (const QDomNotation & x)
- QDomNotation & **operator=** (const QDomNotation & x)
- **~QDomNotation** ()
- QString **publicId** () const
- QString **systemId** () const
- virtual QDomNode::NodeType **nodeType** () const
- virtual bool **isNotation** () const

Detailed Description

The QDomNotation class represents an XML notation.

A notation either declares, by name, the format of an unparsed entity (see section 4.7 of the XML 1.0 specification), or is used for formal declaration of processing instruction targets (see section 2.6 of the XML 1.0 specification).

DOM does not support editing notation nodes; they are therefore read-only.

A notation node does not have any parent.

You can retrieve the publicId() and systemId() from a notation node.

For further information about the Document Object Model see http://www.w3.org/TR/REC-DOM-Level-1/ and http://www.w3.org/TR/DOM-Level-2-Core/. For a more general introduction of the DOM implementation see the QDomDocument documentation.

See also XML.

Member Function Documentation

QDomNotation::QDomNotation ()

Constructor.

QDomNotation::QDomNotation (const QDomNotation & x)

Constructs a copy of *x*.

The data of the copy is shared (shallow copy): modifying one node will also change the other. If you want to make a deep copy, use cloneNode().

QDomNotation::~QDomNotation ()

Destroys the object and frees its resources.

bool QDomNotation::isNotation () const [virtual]

Returns TRUE.

Reimplemented from QDomNode [p. 60].

QDomNode::NodeType QDomNotation::nodeType () const [virtual]

Returns NotationNode.

Reimplemented from QDomNode [p. 62].

QDomNotation & QDomNotation::operator= (const QDomNotation & x)

Assigns *x* to this DOM notation.

The data of the copy is shared (shallow copy): modifying one node will also change the other. If you want to make a deep copy, use cloneNode().

QString QDomNotation::publicId () const

Returns the public identifier of this notation.

QString QDomNotation::systemId () const

Returns the system identifier of this notation.

QDomProcessingInstruction Class Reference

The QDomProcessingInstruction class represents an XML processing instruction.

This class is part of the **XML** module.

`#include <qdom.h>`

Inherits QDomNode [p. 53].

Public Members

- **QDomProcessingInstruction** ()
- **QDomProcessingInstruction** (const QDomProcessingInstruction & x)
- QDomProcessingInstruction & **operator=** (const QDomProcessingInstruction & x)
- **~QDomProcessingInstruction** ()
- virtual QString **target** () const
- virtual QString **data** () const
- virtual void **setData** (const QString & d)
- virtual QDomNode::NodeType **nodeType** () const
- virtual bool **isProcessingInstruction** () const

Detailed Description

The QDomProcessingInstruction class represents an XML processing instruction.

Processing instructions are used in XML as a way to keep processor-specific information in the text of the document.

The content of the processing instruction is retrieved with data() and set with setData(). The processing instruction's target is retrieved with target().

For further information about the Document Object Model see http://www.w3.org/TR/REC-DOM-Level-1/ and http://www.w3.org/TR/DOM-Level-2-Core/. For a more general introduction of the DOM implementation see the QDomDocument documentation.

See also XML.

Member Function Documentation

QDomProcessingInstruction::QDomProcessingInstruction ()

Constructs an empty processing instruction. Use QDomDocument::createProcessingInstruction() to create a processing instruction with content.

QDomProcessingInstruction::QDomProcessingInstruction (const QDomProcessingInstruction & x)

Constructs a copy of *x*.

The data of the copy is shared (shallow copy): modifying one node will also change the other. If you want to make a deep copy, use cloneNode().

QDomProcessingInstruction::~QDomProcessingInstruction ()

Destroys the object and frees its resources.

QString QDomProcessingInstruction::data () const [virtual]

Returns the content of this processing instruction.

See also setData() [p. 73] and target() [p. 74].

bool QDomProcessingInstruction::isProcessingInstruction () const [virtual]

Returns TRUE.

Reimplemented from QDomNode [p. 60].

QDomNode::NodeType QDomProcessingInstruction::nodeType () const [virtual]

Returns ProcessingInstructionNode.

Reimplemented from QDomNode [p. 62].

QDomProcessingInstruction & QDomProcessingInstruction::operator= (const QDomProcessingInstruction & x)

Assigns *x* to this processing instruction.

The data of the copy is shared (shallow copy): modifying one node will also change the other. If you want to make a deep copy, use cloneNode().

void QDomProcessingInstruction::setData (const QString & d) [virtual]

Sets the data contained in the processing instruction to *d*.

See also data() [p. 73].

QString QDomProcessingInstruction::target () const [virtual]

Returns the target of this processing instruction.

See also data() [p. 73].

QDomText Class Reference

The QDomText class represents text data in the parsed XML document.

This class is part of the **XML** module.

`#include <qdom.h>`

Inherits QDomCharacterData [p. 17].

Inherited by QDomCDATASection [p. 15].

Public Members

- **QDomText** ()
- **QDomText** (const QDomText & x)
- QDomText & **operator=** (const QDomText & x)
- **~QDomText** ()
- QDomText **splitText** (int offset)
- virtual QDomNode::NodeType **nodeType** () const
- virtual bool **isText** () const

Detailed Description

The QDomText class represents text data in the parsed XML document.

You can split the text in a QDomText object over two QDomText objecs with splitText().

For further information about the Document Object Model see http://www.w3.org/TR/REC-DOM-Level-1/ and http://www.w3.org/TR/DOM-Level-2-Core/. For a more general introduction of the DOM implementation see the QDomDocument documentation.

See also XML.

Member Function Documentation

QDomText::QDomText ()

Constructs an empty QDomText object.

To construct a QDomText with content, use QDomDocument::createTextNode().

QDomText::QDomText (const QDomText & x)

Constructs a copy of *x*.

The data of the copy is shared (shallow copy): modifying one node will also change the other. If you want to make a deep copy, use cloneNode().

QDomText::~QDomText ()

Destroys the object and frees its resources.

bool QDomText::isText () const [virtual]

Returns TRUE.

Reimplemented from QDomNode [p. 61].

QDomNode::NodeType QDomText::nodeType () const [virtual]

Returns TextNode.

Reimplemented from QDomCharacterData [p. 19].

Reimplemented in QDomCDATASection.

QDomText & QDomText::operator= (const QDomText & x)

Assigns *x* to this DOM text.

The data of the copy is shared (shallow copy): modifying one node will also change the other. If you want to make a deep copy, use cloneNode().

QDomText QDomText::splitText (int offset)

Splits this object at position *offset* into two QDomText objects. The newly created object is inserted into the document tree after this object.

The function returns the newly created object.

See also QDomNode::normalize() [p. 63].

QXmlAttributes Class Reference

The QXmlAttributes class provides XML attributes.

This class is part of the **XML** module.

`#include <qxml.h>`

Public Members

- **QXmlAttributes** ()
- virtual **~QXmlAttributes** ()
- int **index** (const QString & qName) const
- int **index** (const QString & uri, const QString & localPart) const
- int **length** () const
- int **count** () const
- QString **localName** (int index) const
- QString **qName** (int index) const
- QString **uri** (int index) const
- QString **type** (int index) const
- QString **type** (const QString & qName) const
- QString **type** (const QString & uri, const QString & localName) const
- QString **value** (int index) const
- QString **value** (const QString & qName) const
- QString **value** (const QString & uri, const QString & localName) const
- void **clear** ()
- void **append** (const QString & qName, const QString & uri, const QString & localPart, const QString & value)

Detailed Description

The QXmlAttributes class provides XML attributes.

If attributes are reported by QXmlContentHandler::startElement() this class is used to pass the attribute values.

Use index() to locate the position of an attribute in the list, count() to retrieve the number of attributes, and clear() to remove the attributes. New attributes can be added with append(). Use type() to get an attribute's type and value() to get its value. The attribute's name is available from localName() or qName(), and its namespace URI from uri().

See also XML.

Member Function Documentation

QXmlAttributes::QXmlAttributes ()

Constructs an empty attribute list.

QXmlAttributes::~QXmlAttributes () [virtual]

Destroys the attributes object.

void QXmlAttributes::append (const QString & qName, const QString & uri, const QString & localPart, const QString & value)

Appends a new attribute entry to the list of attributes. The qualified name of the attribute is *qName*, the namespae URI is *uri* and the local name is *localPart*. The value of the attribute is *value*.

See also qName() [p. 79], uri() [p. 79], localName() [p. 79] and value() [p. 79].

void QXmlAttributes::clear ()

Clears the list of attributes.

See also append() [p. 78].

int QXmlAttributes::count () const

Returns the number of attributes in the list. This function is equivalent to length().

int QXmlAttributes::index (const QString & qName) const

Looks up the index of an attribute by the qualified name *qName*.

Returns the index of the attribute or -1 if it wasn't found.

See also the namespace description [p. 6].

int QXmlAttributes::index (const QString & uri, const QString & localPart) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Looks up the index of an attribute by a namespace name.

uri specifies the namespace URI, or an empty string if the name has no namespace URI. *localPart* specifies the attribute's local name.

Returns the index of the attribute -1 if it wasn't found.

See also the namespace description [p. 6].

int QXmlAttributes::length () const

Returns the number of attributes in the list.

Example: xml/tagreader-with-features/structureparser.cpp.

QString QXmlAttributes::localName (int index) const

Looks up an attribute's local name for the attribute at position *index*. If no namespace processing is done, the local name is a null string.

See also the namespace description [p. 6].

QString QXmlAttributes::qName (int index) const

Looks up an attribute's XML 1.0 qualified name for the attribute at position *index*.

See also the namespace description [p. 6].

Example: xml/tagreader-with-features/structureparser.cpp.

QString QXmlAttributes::type (int index) const

Looks up an attribute's type for the attribute at position *index*.

Currently only "CDATA" is returned.

QString QXmlAttributes::type (const QString & qName) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Looks up an attribute's type for the qualified name *qName*.

Currently only "CDATA" is returned.

QString QXmlAttributes::type (const QString & uri, const QString & localName) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Looks up an attribute's type by namespace name.

uri specifies the namespace URI and *localName* specifies the local name. If the name has no namespace URI, use an empty string for *uri*.

Currently only "CDATA" is returned.

QString QXmlAttributes::uri (int index) const

Looks up an attribute's namespace URI for the attribute at position *index*. If no namespace processing is done or if the attribute has no namespace, the namespace URI is a null string.

See also the namespace description [p. 6].

Example: xml/tagreader-with-features/structureparser.cpp.

QString QXmlAttributes::value (int index) const

Looks up an attribute's value for the attribute at position *index*.

QString QXmlAttributes::value (const QString & qName) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Looks up an attribute's value for the qualified name *qName*.

See also the namespace description [p. 6].

QString QXmlAttributes::value (const QString & uri, const QString & localName) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Looks up an attribute's value by namespace name.

uri specifies the namespace URI, or an empty string if the name has no namespace URI. *localName* specifies the attribute's local name.

See also the namespace description [p. 6].

QXmlContentHandler Class Reference

The QXmlContentHandler class provides an interface to report the logical content of XML data.

This class is part of the **XML** module.

`#include <qxml.h>`

Inherited by QXmlDefaultHandler [p. 87].

Public Members

- virtual void **setDocumentLocator** (QXmlLocator * locator)
- virtual bool **startDocument** ()
- virtual bool **endDocument** ()
- virtual bool **startPrefixMapping** (const QString & prefix, const QString & uri)
- virtual bool **endPrefixMapping** (const QString & prefix)
- virtual bool **startElement** (const QString & namespaceURI, const QString & localName, const QString & qName, const QXmlAttributes & atts)
- virtual bool **endElement** (const QString & namespaceURI, const QString & localName, const QString & qName)
- virtual bool **characters** (const QString & ch)
- virtual bool **ignorableWhitespace** (const QString & ch)
- virtual bool **processingInstruction** (const QString & target, const QString & data)
- virtual bool **skippedEntity** (const QString & name)
- virtual QString **errorString** ()

Detailed Description

The QXmlContentHandler class provides an interface to report the logical content of XML data.

If the application needs to be informed of basic parsing events, it implements this interface and sets it with QXmlReader::setContentHandler(). The reader reports basic document-related events like the start and end of elements and character data through this interface.

The order of events in this interface is very important, and mirrors the order of information in the document itself. For example, all of an element's content (character data, processing instructions, and/or sub-elements) appears, in order, between the startElement() event and the corresponding endElement() event.

The class QXmlDefaultHandler provides a default implementation for this interface; subclassing from the QXmlDefaultHandler class is very convenient if you only want to be informed of some parsing events.

The startDocument() function is called at the start of the document, and endDocument() is called at the end. Before parsing begins setDocumentLocator() is called. For each element startElement() is called, with endElement() being called at the end of each element. The characters() function is called with chunks of character data; ignorableWhitespace() is called with chunks of whitespace and processingInstruction() is called with processing instructions. If an entity is skipped skippedEntity() is called. At the beginning of prefix-URI scopes startPrefixMapping() is called.

See also the Introduction to SAX2 [p. 5].

See also QXmlDTDHandler [p. 88], QXmlDeclHandler [p. 85], QXmlEntityResolver [p. 90], QXmlErrorHandler [p. 92], QXmlLexicalHandler [p. 97] and XML.

Member Function Documentation

bool QXmlContentHandler::characters (const QString & ch) [virtual]

The reader calls this function when it has parsed a chunk of character data (either normal character data or character data inside a CDATA section; if you have to distinguish between those two types you must use QXmlLexicalHandler::startCDATA() and QXmlLexicalHandler::endCDATA()). The character data is reported in *ch*.

Some readers report whitespace in element content using the ignorableWhitespace() function rather than using this one.

A reader may report the character data of an element in more than one chunk; e.g. a reader might want to report "a<b" in three characters() events ("a ", "<" and " b").

If this function returns FALSE the reader stops parsing and reports an error. The reader uses the function errorString() to get the error message.

bool QXmlContentHandler::endDocument () [virtual]

The reader calls this function after it has finished parsing. It is called just once, and is the last handler function called. It is called after the reader has read all input or has abandoned parsing because of a fatal error.

If this function returns FALSE the reader stops parsing and reports an error. The reader uses the function errorString() to get the error message.

See also startDocument() [p. 83].

bool QXmlContentHandler::endElement (const QString & namespaceURI, const QString & localName, const QString & qName) [virtual]

The reader calls this function when it has parsed an end element tag with the qualified name *qName*, the local name *localName* and the namespace URI *namespaceURI*.

If this function returns FALSE the reader stops parsing and reports an error. The reader uses the function errorString() to get the error message.

See also the namespace description [p. 6].

See also startElement() [p. 84].

Examples: xml/tagreader-with-features/structureparser.cpp and xml/tagreader/structureparser.cpp.

bool QXmlContentHandler::endPrefixMapping (const QString & prefix) [virtual]

The reader calls this function to signal the end of a prefix mapping for the prefix *prefix*.

If this function returns FALSE the reader stops parsing and reports an error. The reader uses the function errorString() to get the error message.

See also the namespace description [p. 6].

See also startPrefixMapping() [p. 84].

QString QXmlContentHandler::errorString () [virtual]

The reader calls this function to get an error string, e.g. if any of the handler functions returns FALSE.

bool QXmlContentHandler::ignorableWhitespace (const QString & ch) [virtual]

Some readers may use this function to report each chunk of whitespace in element content. The whitespace reported in *ch*.

If this function returns FALSE the reader stops parsing and reports an error. The reader uses the function errorString() to get the error message.

bool QXmlContentHandler::processingInstruction (const QString & target, const QString & data) [virtual]

The reader calls this function when it has parsed a processing instruction.

target is the target name of the processing instruction and *data* is the data in the processing instruction.

If this function returns FALSE the reader stops parsing and reports an error. The reader uses the function errorString() to get the error message.

void QXmlContentHandler::setDocumentLocator (QXmlLocator * locator) [virtual]

The reader calls this function before it starts parsing the document. The argument *locator* is a pointer to a QXmlLocator which allows the application to get the parsing position within the document.

Do not destroy the *locator*; it is destroyed when the reader is destroyed (do not use the *locator* after the reader is destroyed).

bool QXmlContentHandler::skippedEntity (const QString & name) [virtual]

Some readers may skip entities if they have not seen the declarations (e.g. because they are in an external DTD). If they do so they report that they skipped the entity called *name* by calling this function.

If this function returns FALSE the reader stops parsing and reports an error. The reader uses the function errorString() to get the error message.

bool QXmlContentHandler::startDocument () [virtual]

The reader calls this function when it starts parsing the document. The reader calls this function just once, after the call to setDocumentLocator(), and before any other functions in this class or in the QXmlDTDHandler class are called.

If this function returns FALSE the reader stops parsing and reports an error. The reader uses the function errorString() to get the error message.

See also endDocument() [p. 82].

Example: xml/tagreader/structureparser.cpp.

bool QXmlContentHandler::startElement (const QString & namespaceURI, const QString & localName, const QString & qName, const QXmlAttributes & atts) [virtual]

The reader calls this function when it has parsed a start element tag.

There is a corresponding endElement() call when the corresponding end element tag is read. The startElement() and endElement() calls are always nested correctly. Empty element tags (e.g. <x/>) cause a startElement() call immediately followed by an endElement() call.

The attribute list provided contains only attributes with explicit values. The attribute list contains attributes used for namespace declaration (i.e. attributes starting with xmlns) only if the namespace-prefix property of the reader is TRUE.

The argument *namespaceURI* is the namespace URI, or a null string if the element has no namespace URI or if no namespace processing is done. *localName* is the local name (without prefix), or a null string if no namespace processing is done, *qName* is the qualified name (with prefix) and *atts* are the attributes attached to the element. If there are no attributes, *atts* is an empty attributes object.

If this function returns FALSE the reader stops parsing and reports an error. The reader uses the function errorString() to get the error message.

See also the namespace description [p. 6].

See also endElement() [p. 82].

Examples: xml/tagreader-with-features/structureparser.cpp and xml/tagreader/structureparser.cpp.

bool QXmlContentHandler::startPrefixMapping (const QString & prefix, const QString & uri) [virtual]

The reader calls this function to signal the begin of a prefix-URI namespace mapping scope. This information is not necessary for normal namespace processing since the reader automatically replaces prefixes for element and attribute names.

Note that startPrefixMapping() and endPrefixMapping() calls are not guaranteed to be properly nested relative to each other: all startPrefixMapping() events occur before the corresponding startElement() event, and all endPrefixMapping() events occur after the corresponding endElement() event, but their order is not otherwise guaranteed.

The argument *prefix* is the namespace prefix being declared and the argument *uri* is the namespace URI the prefix is mapped to.

If this function returns FALSE the reader stops parsing and reports an error. The reader uses the function errorString() to get the error message.

See also the namespace description [p. 6].

See also endPrefixMapping() [p. 82].

QXmlDeclHandler Class Reference

The QXmlDeclHandler class provides an interface to report declaration content of XML data.

This class is part of the **XML** module.

`#include <qxml.h>`

Inherited by QXmlDefaultHandler [p. 87].

Public Members

- virtual bool **attributeDecl** (const QString & eName, const QString & aName, const QString & type, const QString & valueDefault, const QString & value)
- virtual bool **internalEntityDecl** (const QString & name, const QString & value)
- virtual bool **externalEntityDecl** (const QString & name, const QString & publicId, const QString & systemId)
- virtual QString **errorString** ()

Detailed Description

The QXmlDeclHandler class provides an interface to report declaration content of XML data.

You can set the declaration handler with QXmlReader::setDeclHandler().

This interface based upon the SAX2 extension DeclHandler.

The interface provides attributeDecl(), internalEntityDecl() and externalEntityDecl() functions.

See also the Introduction to SAX2 [p. 5].

See also QXmlDTDHandler [p. 88], QXmlContentHandler [p. 81], QXmlEntityResolver [p. 90], QXmlErrorHandler [p. 92], QXmlLexicalHandler [p. 97] and XML.

Member Function Documentation

bool QXmlDeclHandler::attributeDecl (const QString & eName, const QString & aName, const QString & type, const QString & valueDefault, const QString & value) [virtual]

The reader calls this function to report an attribute type declaration. Only the effective (first) declaration for an attribute is reported.

The reader passes the name of the associated element in *eName* and the name of the attribute in *aName*. It passes a string that represents the attribute type in *type* and a string that represents the attribute default in *valueDefault*.

This string is one of "#IMPLIED", "#REQUIRED", "#FIXED" or null (if none of the others applies). The reader passes the attribute's default value in *value*. If no default value is specified in the XML file, *value* is QString::null.

If this function returns FALSE the reader stops parsing and reports an error. The reader uses the function errorString() to get the error message.

QString QXmlDeclHandler::errorString () [virtual]

The reader calls this function to get an error string if any of the handler functions returns FALSE.

bool QXmlDeclHandler::externalEntityDecl (const QString & name, const QString & publicId, const QString & systemId) [virtual]

The reader calls this function to report a parsed external entity declaration. Only the effective (first) declaration for each entity is reported.

The reader passes the name of the entity in *name*, the public identifier in *publicId* and the system identifier in *systemId*. If there is no public identifier specified, it passes QString::null in *publicId*.

If this function returns FALSE the reader stops parsing and reports an error. The reader uses the function errorString() to get the error message.

bool QXmlDeclHandler::internalEntityDecl (const QString & name, const QString & value) [virtual]

The reader calls this function to report an internal entity declaration. Only the effective (first) declaration is reported.

The reader passes the name of the entity in *name* and the value of the entity in *value*.

If this function returns FALSE the reader stops parsing and reports an error. The reader uses the function errorString() to get the error message.

QXmlDefaultHandler Class Reference

The QXmlDefaultHandler class provides a default implementation of all XML handler classes.

This class is part of the **XML** module.

```
#include <qxml.h>
```

Inherits QXmlContentHandler [p. 81], QXmlErrorHandler [p. 92], QXmlDTDHandler [p. 88], QXmlEntityResolver [p. 90], QXmlLexicalHandler [p. 97] and QXmlDeclHandler [p. 85].

Public Members

- **QXmlDefaultHandler** ()
- virtual **~QXmlDefaultHandler** ()

Detailed Description

The QXmlDefaultHandler class provides a default implementation of all XML handler classes.

Very often you are only interested in parts of the things that that the reader reports to you. This class implements a default behaviour for the handler classes (i.e. most of the time do nothing). Usually this is the class you subclass for implementing your customized handler.

See also the Introduction to SAX2 [p. 5].

See also QXmlDTDHandler [p. 88], QXmlDeclHandler [p. 85], QXmlContentHandler [p. 81], QXmlEntityResolver [p. 90], QXmlErrorHandler [p. 92], QXmlLexicalHandler [p. 97] and XML.

Member Function Documentation

QXmlDefaultHandler::QXmlDefaultHandler ()

Constructor.

QXmlDefaultHandler::~QXmlDefaultHandler () [virtual]

Destructor.

QXmlDTDHandler Class Reference

The QXmlDTDHandler class provides an interface to report DTD content of XML data.

This class is part of the **XML** module.

`#include <qxml.h>`

Inherited by QXmlDefaultHandler [p. 87].

Public Members

- virtual bool **notationDecl** (const QString & name, const QString & publicId, const QString & systemId)
- virtual bool **unparsedEntityDecl** (const QString & name, const QString & publicId, const QString & systemId, const QString & notationName)
- virtual QString **errorString** ()

Detailed Description

The QXmlDTDHandler class provides an interface to report DTD content of XML data.

If an application needs information about notations and unparsed entities, it can implement this interface and register an instance with QXmlReader::setDTDHandler().

Note that this interface includes only those DTD events that the XML recommendation requires processors to report, i.e. notation and unparsed entity declarations using notationDecl() and unparsedEntityDecl() respectively.

See also the Introduction to SAX2 [p. 5].

See also QXmlDeclHandler [p. 85], QXmlContentHandler [p. 81], QXmlEntityResolver [p. 90], QXmlErrorHandler [p. 92], QXmlLexicalHandler [p. 97] and XML.

Member Function Documentation

QString QXmlDTDHandler::errorString () [virtual]

The reader calls this function to get an error string if any of the handler functions returns FALSE.

bool QXmlDTDHandler::notationDecl (const QString & name, const QString & publicId, const QString & systemId) [virtual]

The reader calls this function when it has parsed a notation declaration.

The argument *name* is the notation name, *publicId* is the notations's public identifier and *systemId* is the notations's system identifier.

If this function returns FALSE the reader stops parsing and reports an error. The reader uses the function errorString() to get the error message.

bool QXmlDTDHandler::unparsedEntityDecl (const QString & name, const QString & publicId, const QString & systemId, const QString & notationName) [virtual]

The reader calls this function when it finds an unparsed entity declaration.

The argument *name* is the unparsed entity's name, *publicId* is the entity's public identifier, *systemId* is the entity's system identifier and *notationName* is the name of the associated notation.

If this function returns FALSE the reader stops parsing and reports an error. The reader uses the function errorString() to get the error message.

QXmlEntityResolver Class Reference

The QXmlEntityResolver class provides an interface to resolve external entities contained in XML data.

This class is part of the **XML** module.

`#include <qxml.h>`

Inherited by QXmlDefaultHandler [p. 87].

Public Members

- virtual bool **resolveEntity** (const QString & publicId, const QString & systemId, QXmlInputSource *& ret)
- virtual QString **errorString** ()

Detailed Description

The QXmlEntityResolver class provides an interface to resolve external entities contained in XML data.

If an application needs to implement customized handling for external entities, it must implement this interface, i.e. resolveEntity(), and register it with QXmlReader::setEntityResolver().

See also the Introduction to SAX2 [p. 5].

See also QXmlDTDHandler [p. 88], QXmlDeclHandler [p. 85], QXmlContentHandler [p. 81], QXmlErrorHandler [p. 92], QXmlLexicalHandler [p. 97] and XML.

Member Function Documentation

QString QXmlEntityResolver::errorString () [virtual]

The reader calls this function to get an error string if any of the handler functions returns FALSE.

bool QXmlEntityResolver::resolveEntity (const QString & publicId, const QString & systemId, QXmlInputSource *& ret) [virtual]

The reader calls this function before it opens any external entity, except the top-level document entity. The application may request the reader to resolve the entity itself (*ret* is 0) or to use an entirely different input source (*ret* points to the input source).

The reader deletes the input source *ret* when it no longer needs it, so you should allocate it on the heap with `new`.

The argument *publicId* is the public identifier of the external entity, *systemId* is the system identifier of the external entity and *ret* is the return value of this function. If *ret* is 0 the reader should resolve the entity itself, if it is non-zero it must point to an input source which the reader uses instead.

If this function returns FALSE the reader stops parsing and reports an error. The reader uses the function errorString() to get the error message.

QXmlErrorHandler Class Reference

The QXmlErrorHandler class provides an interface to report errors in XML data.

This class is part of the **XML** module.

`#include <qxml.h>`

Inherited by QXmlDefaultHandler [p. 87].

Public Members

- virtual bool **warning** (const QXmlParseException & exception)
- virtual bool **error** (const QXmlParseException & exception)
- virtual bool **fatalError** (const QXmlParseException & exception)
- virtual QString **errorString** ()

Detailed Description

The QXmlErrorHandler class provides an interface to report errors in XML data.

If the application is interested in reporting errors to the user or any other customized error handling, you should subclass this class.

You can set the error handler with QXmlReader::setErrorHandler().

Errors can be reported using warning(), error() and fataError(), with the error text being reported with errorString().

See also the Introduction to SAX2 [p. 5].

See also QXmlDTDHandler [p. 88], QXmlDeclHandler [p. 85], QXmlContentHandler [p. 81], QXmlEntityResolver [p. 90], QXmlLexicalHandler [p. 97] and XML.

Member Function Documentation

bool QXmlErrorHandler::error (const QXmlParseException & exception) [virtual]

A reader might use this function to report a recoverable error. A recoverable error corresponds to the definiton of "error" in section 1.2 of the XML 1.0 specification. Details of the error are stored in *exception*.

The reader must continue to provide normal parsing events after invoking this function.

If this function returns FALSE the reader stops parsing and reports an error. The reader uses the function errorString() to get the error message.

QString QXmlErrorHandler::errorString () [virtual]

The reader calls this function to get an error string if any of the handler functions returns FALSE.

bool QXmlErrorHandler::fatalError (const QXmlParseException & exception) [virtual]

A reader must use this function to report a non-recoverable error. Details of the error are stored in *exception*.

If this function returns TRUE the reader might try to go on parsing and reporting further errors; but no regular parsing events are reported.

bool QXmlErrorHandler::warning (const QXmlParseException & exception) [virtual]

A reader might use this function to report a warning. Warnings are conditions that are not errors or fatal errors as defined by the XML 1.0 specification. Details of the warning are stored in *exception*.

If this function returns FALSE the reader stops parsing and reports an error. The reader uses the function errorString() to get the error message.

QXmlInputSource Class Reference

The QXmlInputSource class provides the input data for the QXmlReader subclasses.

This class is part of the **XML** module.

```
#include <qxml.h>
```

Public Members

- **QXmlInputSource** ()
- **QXmlInputSource** (QIODevice * dev)
- QXmlInputSource (QFile & file) *(obsolete)*
- QXmlInputSource (QTextStream & stream) *(obsolete)*
- virtual **~QXmlInputSource** ()
- virtual void **setData** (const QString & dat)
- virtual void **setData** (const QByteArray & dat)
- virtual void **fetchData** ()
- virtual QString **data** ()
- virtual QChar **next** ()
- virtual void **reset** ()

Protected Members

- virtual QString **fromRawData** (const QByteArray & data, bool beginning = FALSE)

Detailed Description

The QXmlInputSource class provides the input data for the QXmlReader subclasses.

All subclasses of QXmlReader read the input XML document from this class.

This class recognizes the encoding of the data by reading the encoding declaration in the XML file and if it finds one, reading the data using the corresponding encoding. If it does not find an encoding declaration, then it assumes that the data is either in UTF-8 or UTF-16, depending on whether it can find a byte-order mark.

There are two ways to populate the input source with data: you can construct it with a QIODevice* so that the input source reads the data from that device. Or you can set the data explicitly with one of the setData() functions.

Usually you either construct a QXmlInputSource that works on a QIODevice* or you construct an empty QXmlInputSource and set the data with setData(). There are only rare occasions where you want to mix both methods.

The subclasses of QXmlReader use the next() function to read the input character by character. If you want to start from the beginning again, you have to call reset() to change the position in the input source to the beginning.

The functions data() and fetchData() are useful if you want to do something with the data other than parsing, e.g. displaying the raw XML file. The benefit of using the QXmlInputClass in such cases is that it tries to use the correct encoding.

See also QXmlReader [p. 107], QXmlSimpleReader [p. 111] and XML.

Member Function Documentation

QXmlInputSource::QXmlInputSource ()

Constructs an input source which contains no data.

See also setData() [p. 96].

QXmlInputSource::QXmlInputSource (QIODevice * dev)

Constructs an input source and gets the data from device *dev*. If *dev* is not open, it is opened in read-only mode. If *dev* is a null pointer or it is not possible to read from the device, the input source will contain no data.

See also setData() [p. 96], fetchData() [p. 95] and QIODevice [Input/Output and Networking with Qt].

QXmlInputSource::QXmlInputSource (QFile & file)

This function is obsolete. It is provided to keep old source working. We strongly advise against using it in new code.

Constructs an input source and gets the data from the file *file*. If the file cannot be read the input source is empty.

QXmlInputSource::QXmlInputSource (QTextStream & stream)

This function is obsolete. It is provided to keep old source working. We strongly advise against using it in new code.

Constructs an input source and gets the data from the text stream *stream*.

QXmlInputSource::~QXmlInputSource () [virtual]

Destructor.

QString QXmlInputSource::data () [virtual]

Returns the data the input source contains or QString::null if the input source does not contain any data.

See also setData() [p. 96], QXmlInputSource() [p. 95] and fetchData() [p. 95].

void QXmlInputSource::fetchData () [virtual]

This function reads more data from the device that was set during construction. If the input source already contained data, this function deletes that data first.

This object contains no data after a call to this function if the object was constructed without a device to read data from or if this function was not able to get more data from the device.

There are two occasions where a fetch is done implicitly by another function call: during construction (so the object starts out with some initial data where available), and during a call to next() (if the data had run out).

You normally don't need to use this function if you use next().

See also data() [p. 95], next() [p. 96] and QXmlInputSource() [p. 95].

QString QXmlInputSource::fromRawData (const QByteArray & data, bool beginning = FALSE) [virtual protected]

This function reads the XML file from *data* and tries to recoginize the encoding. It converts the raw data *data* into a QString and returns it. It tries its best to get the correct encoding for the XML file.

If *beginning* is TRUE, this function assumes that the data starts at the beginning of a new XML document and looks for an encoding declaration. If *beginning* is FALSE, it converts the raw data using the encoding determined from prior calls.

QChar QXmlInputSource::next () [virtual]

Returns the next character of the input source. If this funciton reaches the end of available data, it returns QXmlInputSource::EndOfData. If you call next() after that, it tries to fetch more data by calling fetchData(). If the fetchData() call results in new data, this function returns the first character of that data; otherwise it returns QXmlInputSource::EndOfDocument.

See also reset() [p. 96], fetchData() [p. 95], QXmlSimpleReader::parse() [p. 112] and QXmlSimpleReader::parseContinue() [p. 112].

void QXmlInputSource::reset () [virtual]

This function sets the position used by next() to the beginning of the data returned by data(). This is useful if you want to use the input source for more than one parse.

See also next() [p. 96].

Example: xml/tagreader-with-features/tagreader.cpp.

void QXmlInputSource::setData (const QString & dat) [virtual]

Sets the data of the input source to *dat*.

If the input source already contains data, this function deletes that data first.

See also data() [p. 95].

void QXmlInputSource::setData (const QByteArray & dat) [virtual]

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

The data *dat* is passed through the correct text-codec, before it is set.

QXmlLexicalHandler Class Reference

The QXmlLexicalHandler class provides an interface to report the lexical content of XML data.

This class is part of the **XML** module.

`#include <qxml.h>`

Inherited by QXmlDefaultHandler [p. 87].

Public Members

- virtual bool **startDTD** (const QString & name, const QString & publicId, const QString & systemId)
- virtual bool **endDTD** ()
- virtual bool **startEntity** (const QString & name)
- virtual bool **endEntity** (const QString & name)
- virtual bool **startCDATA** ()
- virtual bool **endCDATA** ()
- virtual bool **comment** (const QString & ch)
- virtual QString **errorString** ()

Detailed Description

The QXmlLexicalHandler class provides an interface to report the lexical content of XML data.

The events in the lexical handler apply to the entire document, not just to the document element, and all lexical handler events appear between the content handler's startDocument and endDocument events.

You can set the lexical handler with QXmlReader::setLexicalHandler().

This interface's design is based on the the SAX2 extension LexicalHandler.

The interface provides startDTD(), endDTD(), startEntity(), endEntity(), startCDATA(), endCDATA() and comment() functions.

See also the Introduction to SAX2 [p. 5].

See also QXmlDTDHandler [p. 88], QXmlDeclHandler [p. 85], QXmlContentHandler [p. 81], QXmlEntityResolver [p. 90], QXmlErrorHandler [p. 92] and XML.

Member Function Documentation

bool QXmlLexicalHandler::comment (const QString & ch) [virtual]

The reader calls this function to report an XML comment anywhere in the document. It reports the text of the comment in *ch*.

If this function returns FALSE the reader stops parsing and reports an error. The reader uses the function errorString() to get the error message.

bool QXmlLexicalHandler::endCDATA () [virtual]

The reader calls this function to report the end of a CDATA section.

If this function returns FALSE the reader stops parsing and reports an error. The reader uses the function errorString() to get the error message.

See also startCDATA() [p. 98].

bool QXmlLexicalHandler::endDTD () [virtual]

The reader calls this function to report the end of a DTD declaration, if any.

If this function returns FALSE the reader stops parsing and reports an error. The reader uses the function errorString() to get the error message.

See also startDTD() [p. 99].

bool QXmlLexicalHandler::endEntity (const QString & name) [virtual]

The reader calls this function to report the end of an entity with the name *name*.

For every call of startEntity(), there is a corresponding call of endEntity(). The calls of startEntity() and endEntity() are properly nested.

If this function returns FALSE the reader stops parsing and reports an error. The reader uses the function errorString() to get the error message.

See also startEntity() [p. 99] and QXmlSimpleReader::setFeature() [p. 110].

QString QXmlLexicalHandler::errorString () [virtual]

The reader calls this function to get an error string if any of the handler functions returns FALSE.

bool QXmlLexicalHandler::startCDATA () [virtual]

The reader calls this function to report the start of a CDATA section. The content of the CDATA section is reported through the QXmlContentHandler::characters() function. This function is intended only to report the boundary.

If this function returns FALSE the reader stops parsing and reports an error. The reader uses the function errorString() to get the error message.

See also endCDATA() [p. 98].

bool QXmlLexicalHandler::startDTD (const QString & name, const QString & publicId, const QString & systemId) [virtual]

The reader calls this function to report the start of a DTD declaration, if any. It reports the name of the document type in *name*, the public identifier in *publicId* and the system identifier in *systemId*.

If the public identifier and the system identifier is missing, the reader sets the *publicId* and *systemId* to QString::null.

All declarations reported through QXmlDTDHandler or QXmlDeclHandler appear between the startDTD() and endDTD() calls.

If this function returns FALSE the reader stops parsing and reports an error. The reader uses the function errorString() to get the error message.

See also endDTD() [p. 98].

bool QXmlLexicalHandler::startEntity (const QString & name) [virtual]

The reader calls this function to report the start of an entity with the name *name*.

Note that if the entity is unknown, the reader reports it through QXmlContentHandler::skippedEntity() and not throught this function.

If this function returns FALSE the reader stops parsing and reports an error. The reader uses the function errorString() to get the error message.

See also endEntity() [p. 98] and QXmlSimpleReader::setFeature() [p. 110].

QXmlLocator Class Reference

The QXmlLocator class provides the XML handler classes with information about the parsing position within a file.

This class is part of the **XML** module.

```
#include <qxml.h>
```

Public Members

- **QXmlLocator** ()
- virtual **~QXmlLocator** ()
- virtual int **columnNumber** ()
- virtual int **lineNumber** ()

Detailed Description

The QXmlLocator class provides the XML handler classes with information about the parsing position within a file.

The reader reports a QXmlLocator to the content handler before it starts to parse the document. This is done with the QXmlContentHandler::setDocumentLocator() function. The handler classes can now use this locator to get the position (lineNumber() and columnNumber()) that the reader has reached.

See also XML.

Member Function Documentation

QXmlLocator::QXmlLocator ()

Constructor.

QXmlLocator::~QXmlLocator () [virtual]

Destructor.

int QXmlLocator::columnNumber () [virtual]

Returns the column number (starting at 1) or -1 if there is no column number available.

int QXmlLocator::lineNumber () [virtual]

Returns the line number (starting at 1) or -1 if there is no line number available.

QXmlNamespaceSupport Class Reference

This class is part of the **XML** module.

The QXmlNamespaceSupport class is a helper class for XML readers which want to include namespace support.

`#include <qxml.h>`

Public Members

- **QXmlNamespaceSupport** ()
- **~QXmlNamespaceSupport** ()
- void **setPrefix** (const QString & pre, const QString & uri)
- QString **prefix** (const QString & uri) const
- QString **uri** (const QString & prefix) const
- void **splitName** (const QString & qname, QString & prefix, QString & localname) const
- void **processName** (const QString & qname, bool isAttribute, QString & nsuri, QString & localname) const
- QStringList **prefixes** () const
- QStringList **prefixes** (const QString & uri) const
- void **pushContext** ()
- void **popContext** ()
- void **reset** ()

Detailed Description

The QXmlNamespaceSupport class is a helper class for XML readers which want to include namespace support.

You can set the prefix for the current namespace with setPrefix(), and get the list of current prefixes (or those for a given URI) with prefixes(). The namespace URI is available from uri(). Use pushContext() to start a new namespace context, and popContext() to return to the previous namespace context. Use splitName() or processName() to split a name into its prefix and local name.

See also the namespace description [p. 6].

See also XML.

Member Function Documentation

QXmlNamespaceSupport::QXmlNamespaceSupport ()

Constructs a QXmlNamespaceSupport.

QXmlNamespaceSupport::~QXmlNamespaceSupport ()

Destroys a QXmlNamespaceSupport.

void QXmlNamespaceSupport::popContext ()

Reverts to the previous namespace context.

Normally, you should pop the context at the end of each XML element. After popping the context, all namespace prefix mappings that were previously in force are restored.

See also pushContext() [p. 104].

QString QXmlNamespaceSupport::prefix (const QString & uri) const

Returns one of the prefixes mapped to the namespace URI *uri*.

If more than one prefix is currently mapped to the same URI, this function makes an arbitrary selection; if you want all of the prefixes, use prefixes() instead.

Note: to check for a default prefix, use the uri() function with an argument of "".

QStringList QXmlNamespaceSupport::prefixes () const

Returns a list of all prefixes currently declared.

If there is a default prefix, this function does not return it in the list; check for the default prefix using uri() with an argument of "".

QStringList QXmlNamespaceSupport::prefixes (const QString & uri) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns a list of all prefixes currently declared for the namespace URI *uri*.

The "xml:" prefix is included. If you only want one prefix that is mapped to the namespace URI, and you don't care which one you get, use the prefix() function instead.

Note: the empty (default) prefix is never included in this list; to check for the presence of a default namespace, use uri() with an argument of "".

void QXmlNamespaceSupport::processName (const QString & qname, bool isAttribute, QString & nsuri, QString & localname) const

Processes a raw XML 1.0 name in the current context by removing the prefix and looking it up among the prefixes currently declared.

qname is the raw XML 1.0 name to be processed. *isAttribute* is TRUE if the name is an attribute name.

This function stores the namespace URI in *nsuri* (which will get a null string if the raw name has an undeclared prefix), and stores the local name (without prefix) in *localname* (which will get a null string if no namespace is in use).

Note that attribute names are processed differently than element names: an unprefixed element name gets the default namespace (if any), while an unprefixed element name does not.

void QXmlNamespaceSupport::pushContext ()

Starts a new namespace context.

Normally, you should push a new context at the beginning of each XML element: the new context automatically inherits the declarations of its parent context, and it also keeps track of which declarations were made within this context.

See also popContext() [p. 103].

void QXmlNamespaceSupport::reset ()

Resets this namespace support object for reuse.

void QXmlNamespaceSupport::setPrefix (const QString & pre, const QString & uri)

This function declares a prefix *pre* in the current namespace context to be the namespace URI *uri*. The prefix remains in force until this context is popped, unless it is shadowed in a descendant context.

Note that there is an asymmetry in this library. prefix() does not return the default "" prefix, even if you have declared one; to check for a default prefix, you must look it up explicitly using uri(). This asymmetry exists to make it easier to look up prefixes for attribute names, where the default prefix is not allowed.

void QXmlNamespaceSupport::splitName (const QString & qname, QString & prefix, QString & localname) const

Splits the name *qname* at the ':' and returns the prefix in *prefix* and the local name in *localname*.

See also processName() [p. 103].

QString QXmlNamespaceSupport::uri (const QString & prefix) const

Looks up the prefix *prefix* in the current context and returns the currently-mapped namespace URI. Use the empty string ("") for the default namespace.

QXmlParseException Class Reference

The QXmlParseException class is used to report errors with the QXmlErrorHandler interface.

This class is part of the **XML** module.

```
#include <qxml.h>
```

Public Members

- **QXmlParseException** (const QString & name = "", int c = -1, int l = -1, const QString & p = "", const QString & s = "")
- int **columnNumber** () const
- int **lineNumber** () const
- QString **publicId** () const
- QString **systemId** () const
- QString **message** () const

Detailed Description

The QXmlParseException class is used to report errors with the QXmlErrorHandler interface.

The XML subsystem constructs an instance of this class when it detects an error. You can retrieve the place where the error occurred using systemId(), publicId(), lineNumber() and columnNumber(), along with the error message().

See also QXmlErrorHandler [p. 92], QXmlReader [p. 107] and XML.

Member Function Documentation

QXmlParseException::QXmlParseException (const QString & name = "", int c = -1, int l = -1, const QString & p = "", const QString & s = "")

Constructs a parse exception with the error string *name* in the column *c* and line *l* for the public identifier *p* and the system identifier *s*.

int QXmlParseException::columnNumber () const

Returns the column number where the error occurred.

int QXmlParseException::lineNumber () const

Returns the line number where the error occurred.

QString QXmlParseException::message () const

Returns the error message.

QString QXmlParseException::publicId () const

Returns the public identifier where the error occurred.

QString QXmlParseException::systemId () const

Returns the system identifier where the error occurred.

QXmlReader Class Reference

The QXmlReader class provides an interface for XML readers (i.e. parsers).

This class is part of the **XML** module.

`#include <qxml.h>`

Inherited by QXmlSimpleReader [p. 111].

Public Members

- virtual bool **feature** (const QString & name, bool * ok = 0) const
- virtual void **setFeature** (const QString & name, bool value)
- virtual bool **hasFeature** (const QString & name) const
- virtual void * **property** (const QString & name, bool * ok = 0) const
- virtual void **setProperty** (const QString & name, void * value)
- virtual bool **hasProperty** (const QString & name) const
- virtual void **setEntityResolver** (QXmlEntityResolver * handler)
- virtual QXmlEntityResolver * **entityResolver** () const
- virtual void **setDTDHandler** (QXmlDTDHandler * handler)
- virtual QXmlDTDHandler * **DTDHandler** () const
- virtual void **setContentHandler** (QXmlContentHandler * handler)
- virtual QXmlContentHandler * **contentHandler** () const
- virtual void **setErrorHandler** (QXmlErrorHandler * handler)
- virtual QXmlErrorHandler * **errorHandler** () const
- virtual void **setLexicalHandler** (QXmlLexicalHandler * handler)
- virtual QXmlLexicalHandler * **lexicalHandler** () const
- virtual void **setDeclHandler** (QXmlDeclHandler * handler)
- virtual QXmlDeclHandler * **declHandler** () const
- virtual bool parse (const QXmlInputSource & input) *(obsolete)*
- virtual bool **parse** (const QXmlInputSource * input)

Detailed Description

The QXmlReader class provides an interface for XML readers (i.e. parsers).

This abstract class provides an interface for all XML readers in Qt. At the moment there is only one implementation of a reader included in the XML module of Qt (QXmlSimpleReader). In future releases there might be more readers with different properties available (e.g. a validating parser).

The design of the XML classes follows the SAX2 java interface. It was adapted to fit the Qt naming conventions; so it should be very easy for anybody who has worked with SAX2 to get started with the Qt XML classes.

All readers use the class QXmlInputSource to read the input document. Since you are normally interested in particular content in the XML document, the reader reports the content through special handler classes (QXmlDTDHandler, QXmlDeclHandler, QXmlContentHandler, QXmlEntityResolver, QXmlErrorHandler and QXmlLexicalHandler), which you must subclass, if you want to process the contents..

Since the handler classes describe only interfaces you must implement all functions; there is a class (QXmlDefaultHandler) to make this easier; it implements a default behaviour (do nothing) for all functions.

Features and properties of the reader can be set with setFeature() and setProperty respectively. You can set the reader to use your own subclasses with setEntityResolver(), setDTDHandler(), setContentHandler(), setErrorHandler(), setLexicalHandler() and setDeclHandler(). The parse itself is started with a call to parse().

For getting started see also the tiny SAX2 parser walkthrough.

See also QXmlSimpleReader [p. 111] and XML.

Member Function Documentation

QXmlDTDHandler * QXmlReader::DTDHandler () const [virtual]

Returns the DTD handler or 0 if none was set.

See also setDTDHandler() [p. 109].

QXmlContentHandler * QXmlReader::contentHandler () const [virtual]

Returns the content handler or 0 if none was set.

See also setContentHandler() [p. 109].

QXmlDeclHandler * QXmlReader::declHandler () const [virtual]

Returns the declaration handler or 0 if none was set.

See also setDeclHandler() [p. 110].

QXmlEntityResolver * QXmlReader::entityResolver () const [virtual]

Returns the entity resolver or 0 if none was set.

See also setEntityResolver() [p. 110].

QXmlErrorHandler * QXmlReader::errorHandler () const [virtual]

Returns the error handler or 0 if none was set.

See also setErrorHandler() [p. 110].

bool QXmlReader::feature (const QString & name, bool * ok = 0) const [virtual]

If the reader has the feature called *name*, the feature's value is returned. If no such feature exists the return value is undefined.

If *ok* is not 0, then **ok* is set to TRUE if the reader has the feature called *name*; otherwise **ok* is set to FALSE.

See also setFeature() [p. 110] and hasFeature() [p. 109].

bool QXmlReader::hasFeature (const QString & name) const [virtual]

Returns TRUE if the reader has the feature *name*; otherwise returns FALSE.

See also feature() [p. 108] and setFeature() [p. 110].

bool QXmlReader::hasProperty (const QString & name) const [virtual]

Returns TRUE if the reader has the property *name*; otherwise returns FALSE.

See also property() [p. 109] and setProperty() [p. 110].

QXmlLexicalHandler * QXmlReader::lexicalHandler () const [virtual]

Returns the lexical handler or 0 if none was set.

See also setLexicalHandler() [p. 110].

bool QXmlReader::parse (const QXmlInputSource * input) [virtual]

Reads an XML document from *input* and parses it. Returns TRUE if the parsing was successful; otherwise returns FALSE.

Examples: xml/tagreader-with-features/tagreader.cpp and xml/tagreader/tagreader.cpp.

bool QXmlReader::parse (const QXmlInputSource & input) [virtual]

This function is obsolete. It is provided to keep old source working. We strongly advise against using it in new code.

void * QXmlReader::property (const QString & name, bool * ok = 0) const [virtual]

If the reader has the property *name*, this function returns the value of the property and sets **ok* to TRUE; otherwise **ok* is set to FALSE.

See also setProperty() [p. 110] and hasProperty() [p. 109].

void QXmlReader::setContentHandler (QXmlContentHandler * handler) [virtual]

Sets the content handler to *handler*.

See also contentHandler() [p. 108].

Examples: xml/tagreader-with-features/tagreader.cpp and xml/tagreader/tagreader.cpp.

void QXmlReader::setDTDHandler (QXmlDTDHandler * handler) [virtual]

Sets the DTD handler to *handler*.

See also DTDHandler() [p. 108].

void QXmlReader::setDeclHandler (QXmlDeclHandler * handler) [virtual]

Sets the declaration handler to *handler*.

See also declHandler() [p. 108].

void QXmlReader::setEntityResolver (QXmlEntityResolver * handler) [virtual]

Sets the entity resolver to *handler*.

See also entityResolver() [p. 108].

void QXmlReader::setErrorHandler (QXmlErrorHandler * handler) [virtual]

Sets the error handler to *handler*. Clears the error handler if *handler* is 0.

See also errorHandler() [p. 108].

void QXmlReader::setFeature (const QString & name, bool value) [virtual]

Sets the feature called *name* to the given *value*. If the reader doesn't have the feature nothing happens.

See also feature() [p. 108] and hasFeature() [p. 109].

Example: xml/tagreader-with-features/tagreader.cpp.

void QXmlReader::setLexicalHandler (QXmlLexicalHandler * handler) [virtual]

Sets the lexical handler to *handler*.

See also lexicalHandler() [p. 109].

void QXmlReader::setProperty (const QString & name, void * value) [virtual]

Sets the property *name* to *value*. If the reader doesn't have the property nothing happens.

See also property() [p. 109] and hasProperty() [p. 109].

QXmlSimpleReader Class Reference

The QXmlSimpleReader class provides an implementation of a simple XML reader (parser).

This class is part of the **XML** module.

`#include <qxml.h>`

Inherits QXmlReader [p. 107].

Public Members

- **QXmlSimpleReader** ()
- virtual **~QXmlSimpleReader** ()
- virtual bool **parse** (const QXmlInputSource * input, bool incremental)
- virtual bool **parseContinue** ()

Detailed Description

The QXmlSimpleReader class provides an implementation of a simple XML reader (parser).

This XML reader is sufficient for simple parsing tasks. The reader:

- provides a well-formed parser;
- does not parse any external entities;
- can do namespace processing.

Documents are parsed with a call to parse().

See the tiny SAX2 parser walkthrough.

See also XML.

Member Function Documentation

QXmlSimpleReader::QXmlSimpleReader ()

Constructs a simple XML reader with the following feature settings:

- *http://xml.org/sax/features/namespaces* TRUE
- *http://xml.org/sax/features/namespace-prefixes* FALSE
- *http://trolltech.com/xml/features/report-whitespace-only-CharData* TRUE

- *http://trolltech.com/xml/features/report-start-end-entity* FALSE

More information about features can be found in the Qt SAX2 overview.

See also setFeature() [p. 110].

QXmlSimpleReader::~QXmlSimpleReader () [virtual]

Destroys the simple XML reader.

bool QXmlSimpleReader::parse (const QXmlInputSource * input, bool incremental) [virtual]

Reads an XML document from *input* and parses it. Returns FALSE if the parsing detects an error; otherwise returns TRUE.

If *incremental* is TRUE, the parser does not return FALSE when it reaches the end of the *input* without reaching the end of the XML file. Instead it stores the state of the parser so that parsing can be continued at a later stage when more data is available. You can use the function parseContinue() to continue with parsing. This class stores a pointer to the input source *input* and the parseContinue() tries to read from that input souce. This means you should not delete the input source *input* until you've finished your calls to parseContinue(). If you call this function with *incremental* TRUE whilst an incremental parse is in progress a new parsing session will be started and the previous session lost.

If *incremental* is FALSE, this function behaves like the normal parse function, i.e. it returns FALSE when the end of input is reached without reaching the end of the XML file and the parsing can't be continued.

See also parseContinue() [p. 112] and QSocket [Input/Output and Networking with Qt].

bool QXmlSimpleReader::parseContinue () [virtual]

Continues incremental parsing; this function reads the input from the QXmlInputSource that was specified with the last parse() command. To use this function, you *must* have called parse() with the incremental argument set to TRUE.

Returns FALSE if a parsing error occurs; otherwise returns TRUE.

If the input source returns an empty string for the function QXmlInputSource::data(), then this means that the end of the XML file is reached; this is quite important, especially if you want to use the reader to parse more than one XML file.

The case that the end of the XML file is reached without having finished the parsing is not considered as an error — you can continue parsing at a later stage by calling this function again when there is more data available to parse.

This function assumes that the end of the XML document is reached if the QXmlInputSource::next() function returns QXmlInputSource::EndOfDocument. If the parser has not finished parsing when it encounters this symbol, it is an error and FALSE is returned.

See also parse() [p. 112] and QXmlInputSource::next() [p. 96].

Index